TinyML and Efficient Deep Learning Computing(Song Han)

Lee Jun Hyeok (wnsx0000@gmail.com)

April 6, 2025

목차

Part I

서론

1. 서론

1.1. 서론

1.1.1. 서론

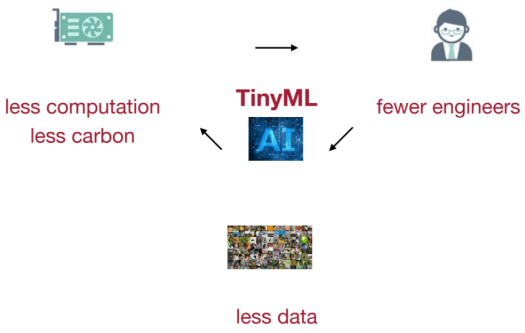
1. 서론

본 필기는 MIT Song Han 교수님의 'TinyML and Efficient Deep Learning Computing(Fall 2024)'을 수강하며 정리했음.

해당 강의는 model compression 등을 통한 edge device에서의 efficient on-device DL을 주된 내용으로 함. edge device에서는 cloud에 비해 성능과 자원 등에서의 제약이 엄격한데, 특히 스마트폰(mobile ai)에서 IOT device(tiny ai)로 갈수록 그 정도가 심해지므로 적절한 최적화가 필수적임.

TinyML은 edge에서 on-device로 실행할 수 있도록 최적화된 model 또는 그런 기법을 말함. TinyML의 동작을 위해서는 적은 연산량과 메모리 사용, 데이터, 인력 등의 요구사항을 만족시켜야 함.

필기의 각 내용은 대응되는 논문을 기반으로 하고 있으므로 구체적인 사항이 궁금하다면 관련 논문을 읽어보자.



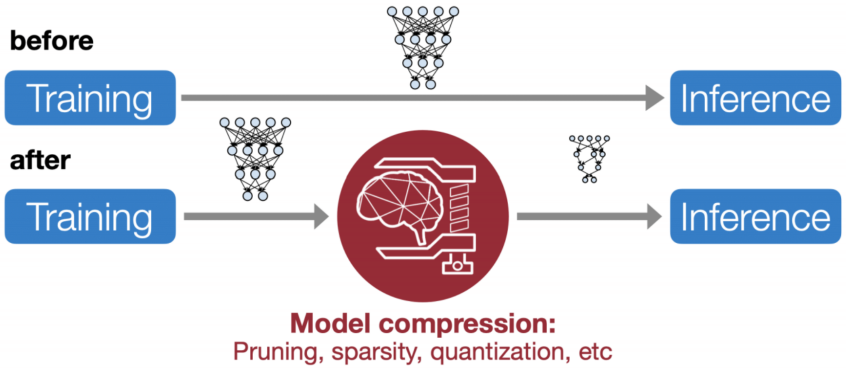
2. 내용 구성

이 수업에서 다루는 내용은 아래와 같음.

1) model compression(경량화)을 통한 efficient inference(추론). pruning, quantization, neural architecture search, distillation 등.

학습 이후 model compression을 거쳐 성능을 유지 또는 개선하면서 model을 최적화함. 다양한 곳에 DL이 활용되고 있지만 이는 많은 cost를 요구하므로 model compression이 중요함.

2015년 즈음부터 model compression 관련 출판의 수가 급격하게 증가하고 있고, 특히 최근 들어서는 model size가 하드웨어적인 지원 가능 범위 이상으로 커짐에 따라 이 간극을 model compression을 통해 좁힐 필요가 있음.



2) efficient training. gradient compression, on-device training, federated learning 등.

3) application-specific optimizations. 1, 2번에서 다룬 efficient DL을 바탕으로 LLM, AIGC(AI Generated Content), 비디오 등에 대한 최적화를 다룸.

1.1.2. 워크로드별 개요

DL의 각 워크로드별로 어떤 방식의 최적화가 가능한지 개괄적으로 알아보자.

DL에 대한 3가지 핵심 구성 요소는 알고리즘, 하드웨어, 데이터임. 각각에 대한 최적화와 개선을 고려할 수 있음.

1. Vision

2010년 이래로 AlexNet, GoogleNet, ResNet 등이 등장하며 최근 들어서는 DL model이 인간의 인식 수준을 넘어서는 성능을 가지게 되었지만, 높은 accuracy를 가지는 model들은 그만큼 많은 연산과 메모리가 필요함. efficient DL은 accuracy를 유지하거나 높이면서 cost를 줄일 수 있도록 함.

efficient DL을 통한 최적화는 연산량과 메모리 사용을 줄여서, DL model이 스마트폰이나 IOT device(스마트폰같은 기기보다 더 제한됨.)같은 edge device에서 cloud server와의 통신 없이 on-device로 동작할 수 있도록 함.

특히 ai 시스템은 계속해서 새로운 입력 데이터를 학습하고 적응해야 하는데, 학습 시에는 각 상태를 저장하고 backpropagation 연산 등을 수행해야 하므로 inference보다 비용이 더 큼. efficient DL에 의해 구현될 수 있는, device에서 자체적으로 수행되는 On-device learning은 privacy, cost, custmization, life-long learning 등의 이점을 가짐.

efficient DL이 충분히 효율적이면 classification뿐만 아니라 detection, image/video generation 등도 제한된 환경에서 수행이 가능함.

2. Language

LLM, code generation, translation 등은 많은 연산량을 필요로 하고, 또 그 model의 크기도 점점 커지고 있음. efficient DL을 통해 이를 개선할 수 있음.

특히 LLM을 pc, 이동수단, 로봇 등에서 on device로 돌리는 것은 latency, 인터넷 연결, privacy 등의 측면에서 이점이 많음.

LLM에는 redundancy가 굉장히 많으므로 정답에 영향이 없거나 적은 일부 token을 pruning할 수 있음.

또한 SmoothQuant, AWQ 등을 활용한 LLM quantization을 수행할 수 있음.

3. Multimodal
Multimodal은 여러 종류의 modality를 활용하는 DL 분야를 말함. 예를 들어, 이미지 데이터와 텍스트 데이터를 함께 활용할 수 있음.

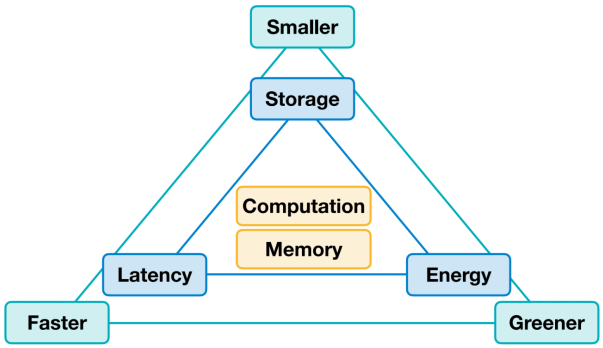
이때 중요한 것은 각 modality의 데이터에 대한 tokenize를 어떻게 수행할 것인가임. 모든 데이터는 token으로 변환할 수 있음.

2. NN Basics

2.1. Efficiency Metrics

2.1.1. Efficiency in NN

1. Efficiency in NN
NN에서의 Efficiency(효율)는 아래와 같이 storage, latency, energy의 측면에서 생각할 수 있음. 모델은 더 작을수록, 빠를수록, 에너지를 덜 사용할수록 efficiency가 높다고 할 수 있음.



각 측면은 Computation(계산)과 Memory를 고려하여 살펴볼 수 있음. 아래에 정리한 NN에서 efficiency를 나타낼 때 사용하는 Metric(평가지표)는 memory와 관련된 것과 computation과 관련된 것으로 나뉨.

2. Latency
Latency는 특정 작업을 완료하는 데에 걸리는 시간임. 예를 들어, model에서는 예측에 걸리는 시간이고, 영상에서는 각 프레임 별 처리에 걸리는 시간임.

latency는 아래와 같이 computation 처리 시간과 memory 처리 시간 중 큰 것에 따라 결정됨. 이때 memory 처리 시간에는 activation도 고려해야 하는데, activation을 활용하려면 그만큼의 memory 공간이 필요하기 때문임.

이때 memory bandwidth는 memory의 데이터 이동 속도임. bandwidth는 분야에 따라 데이터 이동 속도, 주파수 범위, 처리 가능한 전체 데이터량 등 여러 의미를 가짐.

$$Latency \approx \max\left(T_{computation}, T_{memory}\right)$$

$$T_{\text{computation}} \approx \frac{\text{Number of Operations in Neural Network Model}}{\text{Number of Operations that Processor can Process Per Second}}$$

NN Specification / Hardware Specification

$$T_{\text{memory}} \approx T_{\text{data movement of activations}} + T_{\text{data movement of weights}}$$

$$T_{\text{data movement of weights}} \approx \frac{\text{Neural Network Model Size}}{\text{Memory Bandwidth of Processor}}$$

NN Specification / Hardware Specification

$$T_{\text{data movement of activations}} \approx \frac{\text{Input Activation Size + Output Activation Size}}{\text{Memory Bandwidth of Processor}}$$

NN Specification / Hardware Specification

3. Throughput

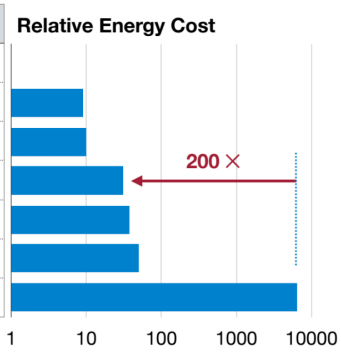
Throughput(처리량)은 데이터 처리 속도(rate)임. 즉, 시간 당 처리한 데이터의 개수임. 예를 들어, 영상에서 각 프레임을 단위로 한다면 초당 처리한 프레임의 개수임.

쉽게 오해할 수 있지만, latency와 throughput은 반비례 관계에 있지 않음. 특히 병렬 처리 시스템 등에서는 이 둘이 모두 높거나 낮을 수 있음. 또한 모바일 기기에서는 latency를, 데이터센터 등에서는 throughput을 주로 신경쓰는 등 환경에 따라 중요한 것이 다름.

4. Energe Comsumption
프로세서의 instruction 종류에 따라 에너지 소비량이 다름. 프로세서로부터 멀리 떨어진 device(register, cache, memory 등)일수록 더 많은 에너지가 필요함.

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	640

Rough Energy Cost For Various Operations in 45nm 0.9V



참고로, NN의 width는 성능에 영향이 큼. 동일한 개수의 parameter를 가진다고 했을 때 wide-shallow한 NN은 gpu를 활용해 효율적으로 병렬 처리를 할 수 있으므로 하드웨어적으로 효율적이고, kernal cause(kernel에서의 병목)가 적고, 역전파에 의한 연산이 적음. 반면에 narrow-deep한 NN은 더 좋은 accuracy를 보임. 이 둘은 tradeoff 관계에 있으므로 적정한 NN을 구성하는 것이 중요함.

2.1.2. Memory-related Metrics

NN의 efficiency를 나타내는 metric 중 memory와 관련된 것을 알아보자. 사용 가능한 리소스보다 큰 memory를 필요로 할수록 성능이 더 떨어짐.

1. Paramerter의 개수
parameter의 개수에 따라 memory를 사용해야 하므로 parameter의 개수는 metric으로 활용됨.

parameter의 개수는 아래와 같이 layer의 종류별로 다르게 계산됨.

1) linear layer : W의 원소의 개수. 입력 channel(벡터의 크기)을 c_i, 출력 channel을 c_o라 하면 $c_i \cdot c_o$임.

2) convolution layer : filter의 원소의 개수. 입력 channel(벡터의 크기)을 c_i, 출력 channel을 c_o, filter의 세로와 가로를 각각 k_h, k_w라 하면 $c_i \cdot c_o \cdot k_h \cdot k_w$ 임.

3) grouped convolution layer : convolution과 같은데 group을 적용한 것이므로, group의 개수를 g라 하면, $c_i/g \cdot c_o/g \cdot k_h \cdot k_w \cdot g = c_i \cdot c_o \cdot k_h \cdot k_w/g$임.

4) depthwise convolution layer : grouped convolution과 같은데, $g = c_i$이므로 $c_o \cdot k_h \cdot k_w$ 임.

2. Model Size
Model Size는 해당 model이 가진 weight를 저장하는 데에 필요한 전체 memroy 크기임. 즉, 아래의 수식으로 계산할 수 있음. bit width는 하나의 parameter가 차지하는 memory 크기인데, 일반적인 경우 NN의 모든 parameter는 동일한 크기를 사용함.

$$Model\ Size = \#Parameters \cdot Bit\ Width$$

parameter에는 puning을, bit width는 quantization을 적용하여 model size를 줄임.

3. Activation의 개수
각 layer의 출력값인 activation이 가지는 원소의 개수에 따라 memory를 사용해야 하므로 activation

의 개수는 metric으로 활용됨. activation의 개수는 아래와 같이 total activation과 peak activation으로 생각할 수 있음.

1) Total Activation은 전체 memory 사용량을 나타냄. 각 layer의 출력 tensor 크기를 전부 더해 계산할 수 있음.

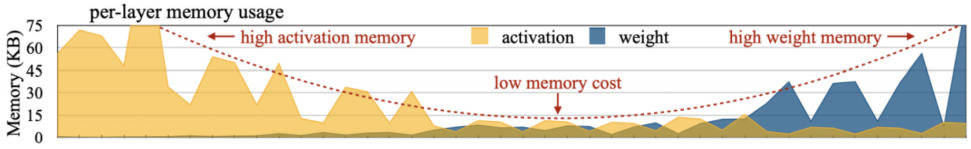
2) Peak Activation은 필요한 최대 memory를 나타냄. 각 layer 중 input activation과 output activation의 크기를 더해 계산할 수 있음.

AlexNet	C × H × W	
Image (3×224×224)	3×224×224	=150,528
11×11 Conv, channel 96, stride 4, pad 2	96×55×55	=290,400
3×3 MaxPool, stride 2	96×27×27	=69,984
5×5 Conv, channel 256, pad 2, groups 2	256×27×27	=186,624
3×3 MaxPool, stride 2	256×13×13	=43,264
3×3 Conv, channel 384, pad 1	384×13×13	=64,896
3×3 Conv, channel 384, pad 1, groups 2	384×13×13	=64,896
3×3 Conv, channel 256, pad 1, groups 2	256×13×13	=43,264
3×3 MaxPool, stride 2	256×6×6	=9,216
Linear, channel 4096	4096	=4,096
Linear, channel 4096	4096	=4,096
Linear, channel 1000	1000	=1,000

Total #Activation: 932,264
Peak #Activation:
≈ #input activation + #output activation
= 150,528 + 290,400 = 440,928

CNN에서 memory 병목을 유발하는 것은 parameter가 아니라 activation임. infernece의 경우 activation의 개수를 줄이기가 어렵고, training의 경우 gpu 메모리를 많이 먹음.

또한 CNN에서는 weight와 activation의 memory 사용 분포가 서로 다르고, 각각 imbalance해서 최적화 방식이 다름. 초반에는 높은 해상도 때문에 activation의 사용량이 높고, 후반에는 많은 channel에 의해 weight의 사용량이 높음.



2.1.3. Computation-related Metrics

NN의 efficiency를 나타내는 metric 중 computation과 관련된 것을 알아보자.

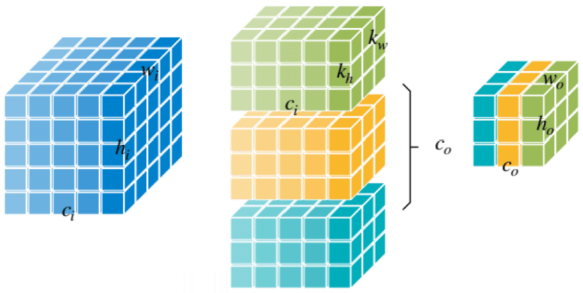
1. MAC

MAC(Multiply-Accumulate Operation)는 덧셈 연산 한 번과 곱셈 연산 한 번을 묶은 연산임. 이는 행렬 곱의 기본 연산이므로 gpu에서 주로 하나의 instruction으로 제공됨. MAC의 개수로 conputation의 양을 파악할 수 있음.

$m \times k$인 행렬과 $k \times n$인 행렬을 곱하는 경우 MAC의 개수는 $m \cdot k \cdot n$임을 생각하면, 아래와 같이 layer의 종류별로 MAC을 계산할 수 있음.

1) linear layer : 단순 행렬 곱이므로, 입력 channel(벡터의 크기)을 c_i, 출력 channel을 c_o라 하면 $c_i \cdot c_o$임.

2) convolution layer : 출력 tensor의 각 원소별로 filter 하나가 연산에 사용되므로, 입력 channel(벡터의 크기)을 c_i, 출력 channel을 c_o, filter의 세로와 가로를 각각 k_h, k_w, 출력 tensor의 세로와 가로를 각각 h_o, w_o라 하면 $c_i \cdot k_h \cdot k_w \cdot h_o \cdot w_o \cdot c_o$임.



3) grouped convolution layer : convolution과 같은데 group을 적용한 것이므로, group의 개수를 g라 하면, $c_i/g \cdot k_h \cdot k_w \cdot h_o \cdot w_o \cdot c_o$임.

4) depthwise convolution layer : grouped convolution과 같은데, $g = c_i$이므로 $k_h \cdot k_w \cdot h_o \cdot w_o \cdot c_o$임.

2. FLOP/FLOPS
FLOP(Floating Poing Operation)은 부동소수점 연산임. 데이터가 부동소수점으로 저장된 경우(실수) 덧셈과 곱셈 연산 각각이 부동소수점 연산이므로, FLOP의 개수는 MAC 개수의 2배임.

FLOPS(FLOPS per Second)는 초당 처리된 FLOP의 개수임.

3. OP/OPS
OP(Operation)은 모든 연산 가리킴. NN의 각 데이터가 부동소수점으로 저장되어 있지 않은 경우도 존재하므로 이를 일반화한 것임.

OPS(OPs per Second)는 초당 처리된 OP의 개수임.

GOPs(Giga OPs per Second)는 초당 수행할 수 있는 연산의 개수를 기가(G) 단위로 나타낸 것임.

OPs/byte는 1바이트의 데이터 당 사용되는 연산의 개수임. 이 값이 클수록 memory에 비해 computation이 많이 필요한 연산인데, 대부분의 환경에서는 memory보다 computation이 싸기 때문에 OPs/byte 값이 큰 것이 이상적임.

Part II

Efficient Inference

1. Pruning/Sparsity

1.1. Pruning/Sparsity

1.1.1. Pruning/Sparsity

1. Pruning
Pruning(가지치기)은 dense한 NN의 redundant(불필요한) weight/activation 등을 가지치기하여 기존 성능을 유지하거나 개선하면서 sparse하고 더 작은 NN으로 변환하는 기법임. 이때 어떤 대상을 prune한다는 것은 해당 값을 0으로 바꾸거나 사용하지 않는 것임.

NN에서 pruning은 weight(synapse, filter), activation(neuron, channel)에 대해 적용할 수 있음. 이때 neuron은 synapse의 연결로 구성되므로 activation pruning은 weight에 대한 coarse-grained pruning과 동일함. 예를 들어, linear layer에서는 weight의 한 행을 제거하는 것이 neuron pruning이고, convolution layer에서는 filter의 특정 부분을 제거하는 것이 channel(neuron) pruning임.

이를 수학적으로 나타내면 아래와 같음. 이때 $argmin_x(f(x))$(선형대수학에서의 표현임.)은 f(x)가 최소가 되도록 하는 x값임. L은 cost function, x는 입력, W_P는 pruned weight, N은 0이 아닌 값의 목표

개수임. $||W_p||_0$는 W_p의 L0 norm(엄밀하게는 노름이 아님.)으로, W_p의 0이 아닌 원소의 개수임. 즉, pruning은 W_p가 0이 아닌 값을 N개 미만으로 가질 때 cost가 최소가 되도록 하는 W_p를 찾는 기법임.

$$arg\min_{W_p} L(x; W_p) \;\; s.t. ||W_P||_0 < N$$

memory 사용은 연산 시간, 에너지, 공간 측면에서 비용이 많이 듦. pruning은 weight와 activation을 줄여 efficiency를 높이는데, 특히 하드웨어가 아닌 알고리즘의 개선으로 efficiency를 확보한다는 점에서 효용이 있음.

이때 어떤 형태로 pruning하는지(granularity), 어떤 기준으로 pruning하는지(criterion), 얼마나 pruning하는지(ratio)는 뒤에서 살펴보자.

2. Sparsity
Sparsity(희소성)는 전체 원소의 개수에 대한 값이 0인 원소의 비율임. pruning을 적용하면 sparsity는 높아짐.

$$Sparsity = 1 - \frac{||W||_0}{Total\ elements\ in\ W}$$

실제 사람의 뇌에서도 pruning이 일어난다고 함.

1.2. Pruning Granularity

1.2.1. Pruning Granularity

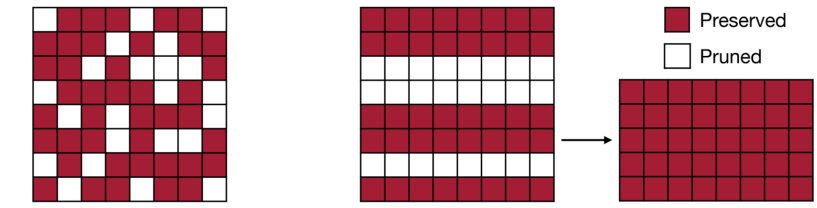
Pruning Granularity(세분성)는 pruning에서의 제거 단위임. 즉, pruning을 적용하는 패턴을 말함.

granularity가 irregular할수록 flexible하고, 이에 따라 redundant한 요소를 개별적으로 찾을 수 있어 pruning ratio가 높음. 반면 regular할수록 규칙적이어서 하드웨어 친화적이고, 병렬 처리가 쉽고, acceleration에 유리함. 이런 각 이점은 tradeoff 관계에 있음.

linear layer의 weight와 convolution layer의 filter(weight) 각각에 대해 어떤 granularity로 pruning할 수 있는지, 각각의 tradeoff는 무엇인지 알아보자.

1.2.2. Linear Layer에 대한 Granularity

W가 matrix인 linear layer의 경우 아래와 같은 granularity가 있음.

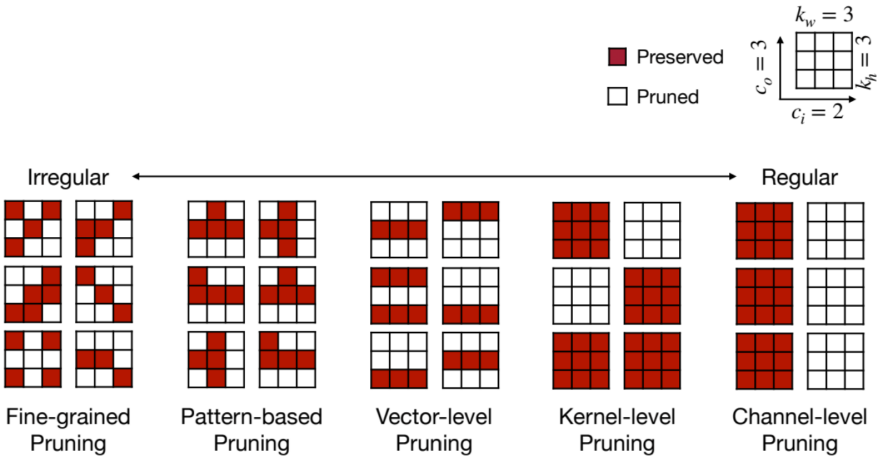


1. Fine-grained Pruning
Fine-grained/Unstructed Pruning은 W의 각 요소에 대해 임의의 위치에 있는 값을 pruning하는 방식임.

2. Coarse-grained Pruning
Coarse-grained/Structed Pruning은 특정 행 또는 열 전체를 pruning하는 방식임.

1.2.3. Convolution Layer에 대한 Granularity

filter(W)가 4차원 tensor인 convolution layer의 경우 아래와 같은 granularity가 있음. 2차원 tensor인 linear layer의 W보다 더 다양한 방식이 존재함. 아래 그림에 있는 각 기법은 동일한 맥락의 tradeoff를 가지므로 전부 정리하지는 않음.



1. Fine-grained Pruning

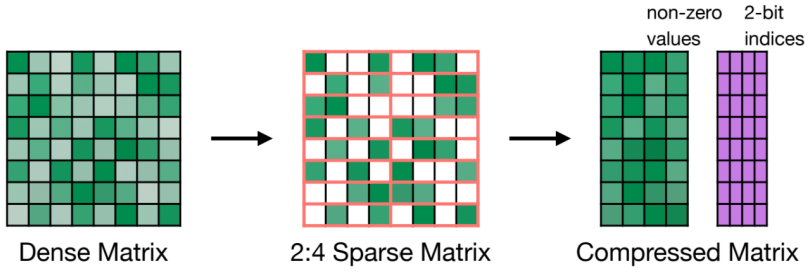
Fine-grained/Unstructed Pruning은 앞에서 다룬 것과 같이 W의 각 요소에 대해 임의의 위치에 있는 값을 pruning하는 방식임.

acceleration, parellelism보다 flexibility가 중요하다면 선택할 수 있음. 이는 특수한 하드웨어에서는 성능이 좋을 수 있지만 shelf hardware(범용 하드웨어)에 대해서는 쉽게 accelerate할 수 없음.

2. Pattern-based Pruning

Pattern-based Pruning은 특정 패턴에 일치하는 위치에 있는 값을 pruning하는 방식임.

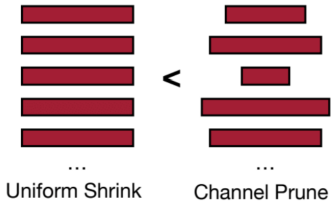
N:M Sparsity는 M개의 요소 중에서 N개를 prune하는 pattern-based pruning임. 예를 들어, 2:4는 4개중 2개를 삭제하여 50%의 sparsity를 확보함. 또한 각 부분에 대해 어느 위치에 0 또는 0이 아닌 원소가 들어있는지를 나타내는 메타데이터 matrix를 생성하여 쉽게 처리할 수 있음. N:M은 대체로 accuracy를 잘 유지한다고 함.

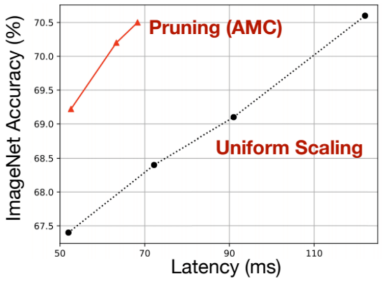


3. Channel-level Pruning

Channel-level Pruning은 특정 channel 전체를 pruning하는 방식임.

여러 layer에 pruning을 적용한다고 했을 때, 각 layer를 모두 동일한 pruning ratio(sparsity)로 pruning할 수도 있고, 개별적으로 최적화된 ratio로 pruning할 수도 있음. 후자가 대체로 더 좋음. 예를 들어, 동일한 구조의 transformer를 여러 layer로 쌓아서 사용하는 model이 있다고 해보자. 물론 model 수준에서는 partition하기도 쉽고 여러 gpu를 사용한 병렬 처리도 간단하지만, 실질적으로 필요한 정도의 pruning이 다르기 때문에 efficieny를 더 올리려면 각 layer별 최적화가 필요함.





1.3. Pruning Criterion

1.3.1. Pruning Criterion

Pruning Criterion은 pruning에 대한 기준임. 즉, 어떤 것을 삭제할 것인지에 대한 기준임.

기본적으로 synpase/neuron에서 덜 중요한 것을 선택하여 삭제해야 함. 덜 중요한 것들을 제거할수록 NN의 efficiency는 높아짐. pruning criterion은 각 대상에 대한 중요도를 파악하는 기준으로 생각할 수 있음.

이때 실질적으로 중요도를 완벽하게 파악할 수는 없으므로 heuristic 알고리즘들을 사용함. 또한 데이터셋, NN 등에 따라 어떤 기법이 좋은지는 다를 수 있기 때문에, 어떤 기법을 적용할지는 실험적으로 테스트하여 결정하게 됨.

Heuristic(휴리스틱, 발견적) 알고리즘이란 특정 문제에 대한 근사적인 해결책을 찾기 위한 경험적이고 규칙적인 방법임. 불충분한 시간이나 정보로 인하여 합리적인 판단을 할 수 없거나, 체계적이면서 합리적인 판단이 굳이 필요하지 않은 상황에서 사람들이 빠르게 사용할 수 있도록 구성된 추론 방법인 것.

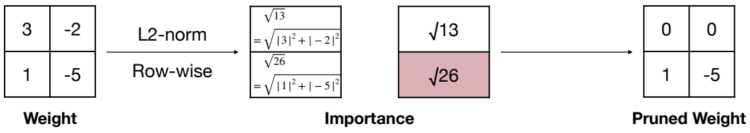
1.3.2. Weight Pruning Criterion

1. Magnitude-based Pruning

Magnitude-based Pruning은 크기를 중요도로 하는 방식임. 이는 다양한 granularity에 대한 pruning에 활용할 수 있음.

fine-grained에 대해서는 각 요소별로 확인하여 절댓값이 작은 것을 삭제함. coarse-grained 등 여러 요소들을 고려하는 경우 해당 요소들에 대한 L1-norm을 계산하여 비교함. 또는 L2-norm, Lp-norm 등을 적용할 수도 있음. p값이 커질수록 해당 부분에서 큰 값의 비중이 커짐.

간단하면서도 잘 작동한다고 함.

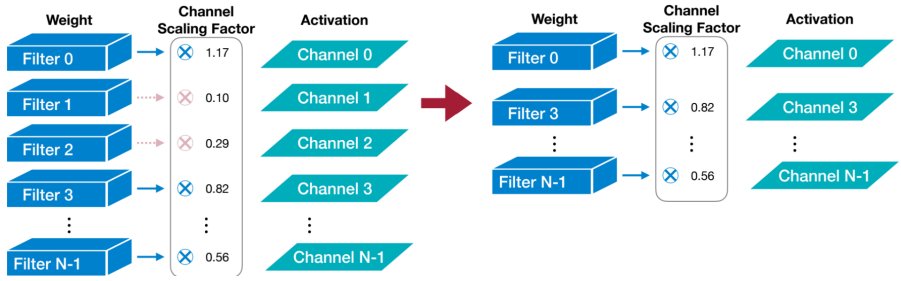


2. Scaling-based Pruning

Scaling-based Pruning은 convolution layer의 각 filter를 거친 출력값에 중요도를 나타내는 scaling factor를 곱하는 방식임. 이를 통해 channel-level pruning을 수행할 수 있음.

scaling factor는 parameter로, channel별로 하나의 값이 존재함. 이는 backpropagation으로 최적화됨. 이후 그 값이 작은 scaling factor에 해당되는 channel은 삭제됨.

이때의 scaling factor는 batch normalization에서 재사용이 가능함. 즉, batch normalization에서 scaling을 위해 사용한 γ로 사용할 수 있음.



3. Second-order-based Pruning

Second-order-based Pruning은 hessian matrix 값을 활용해 각 가중치의 중요도를 계산하는 방식임.

h_{ii}는 아래와 같이 계산되는 hessian matrix H의 원소임. Hessian Matrix는 어떤 다변수함수에 대한 이계도함수를 원소로 가지는 행렬임. 근데 hessian matrix 값은 연산이 복잡해 직접 계산하기보다는 근사해서 값을 얻는다고 함.

$$h_{ij} = \frac{\partial^2 L}{\partial w_i \partial w_j}$$

테일러 전개를 사용하면 기존 model의 loss와 pruning을 적용한 model의 loss간의 차이는 아래와 같이 근사할 수 있음. 계산 과정이 궁금하다면 Lec03-Pruning-1의 p62를 보자. 두 버전 사이의 loss 차이가 클수록 중요도가 높은 것임. 즉, 각 가중치에 대해 아래의 수식을 계산해 중요도를 얻고, 그 값이 작은 것을 삭제함.

$$Importance = |\delta L_i| = \frac{1}{2} h_{ii} w_i^2$$

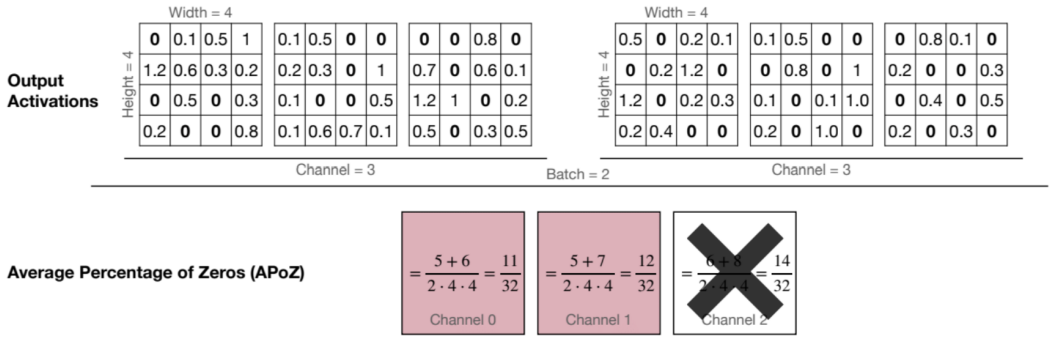
1.3.3. Activation Pruning Criterion

1. Percentage-of-zero-based Pruning

Percentage-of-zero-based Pruning은 0의 개수로 중요도를 계산하는 방식임. 이를 통해 activation에 대해 channel-level pruning을 수행할 수 있음.

ReLU를 사용하는 convolution layer에서, 각 channel에 대해 ReLU에 의해 activation으로 도출된 0의 개수를 세서 그 개수가 가장 많은 channel을 제거함. 이때 전체 요소에 대한 0의 비율을 APoZ(Average Percentage of Zeros)라고 함.

weight에 대한 pruning의 경우 static(정적)으로 처리가 가능했지만, 이 방식은 activation을 활용하므로 sample을 입력해 동적으로 값을 확인해야 함.



2. Regression-based Pruning

Regression-based Pruning은 reconstruction error를 반복 계산하고 최적화하여 construction error가 최소가 되는 W와 β를 찾아, weight에서 특정 channel을 삭제하는 방식임. 이를 통해 coarse-grained pruning을 수행할 수 있음.

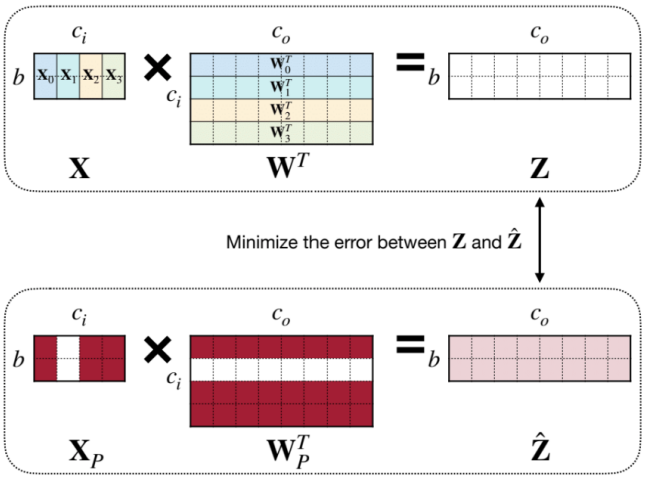
end to end로 NN을 돌리는 것은 비싸기 때문에 regression-based pruning에서는 model 전체를 사용해 loss를 계산하여 pruning하는 대신, layer 각각(layer-wise)에 대한 reconstruction error를 계산하여 pruning함. 즉, 하나의 행렬 곱에 대한 처리를 함. 이때 Reconstruction Error(재구성 오차)는 기존 원본 데이터와, 원본 데이터로 재구성(reconstruct)한 데이터 사이의 오차(error)를 말함. 이를 통해 전체 model을 거치며 backpropagation을 수행하는 대신 최적화를 쉽고 local하게 할 수 있음. 특히 LLM 등 backpropagtation에 대한 비용이 큰 model의 경우 그 효용이 더 커짐.

이를 수식적으로 나타내면 아래와 같음. X_c, W_c는 아래 그림과 같고, N_c는 0이 아닌 channel의 목표 개수임. β는 channel 선택에 대한 계수로, 특정 channel에 대해 값이 0이면 삭제된 것임. $||Z - \hat{Z}||_F$는 Frobenius Norm으로, 행렬의 모든 원소를 제곱한 값을 합한 다음 제곱근을 취한 값임. 이때 실제로 값을 찾을 때는 W를 고정하고 error가 최소화되는 β를 우선 찾은 뒤, β를 고정하고 error가 최소화되는 W를 찾는 작업을 반복 수행함.

$$Z = XW^T = \sum_{c=0}^{c_i-1} X_c W_c^T$$

$$arg\min_{W,\beta} ||Z - \hat{Z}||_F^2 = ||Z - \sum_{c=0}^{c_i-1} \beta_c X_c W_c^T||_F^2 \qquad s.t. ||\beta||_0 \leq N_c$$

이렇게 pruning을 적용해도 출력의 shape은 동일함.



1.4. Pruning Ratio

1.4.1. Pruning Ratio

Pruning Ratio는 전체 weight/activation에 대해 삭제된 대상의 비율임. pruning ratio를 지정하여 얼마나 삭제할지에 대한 기준을 세움.

pruning ratio를 찾는 방법에는 sensitivity analysis, AMC, NetAdapt 등이 있음. 이때 앞에서 다룬 것처럼 pruning 시에는 layer 별로 최적화된 pruning ratio를 찾아야 함.

1.4.2. Sensitivity Analysis

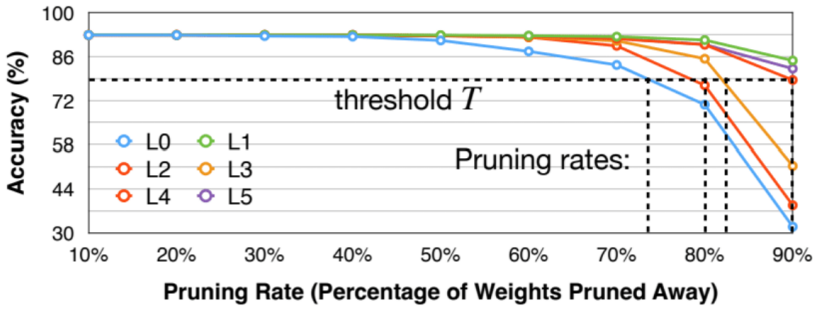
Sensitiviy Analysis는 각 layer별 sensitivity를 고려하여 pruning ratio를 찾는 방식임.

Sensitivity는 pruning ratio에 따른 accuracy 변화 감도를 말함. sensitivity가 높으면 pruning할수록 accuracy가 빠르게 줄어듦.

sensitivity analysis는 아래의 과정을 거쳐 수행됨.

1) 각 layer에 대해 개별적으로 여러 pruning rate를 적용했을 때의 accuracry를 계산하여 sensitivity를 확인함.

2) 요구사항에 부합하는 accuracy threshold를 지정해, 각 layer별로 해당 accuaracy에 맞는 pruning ratio를 찾음.



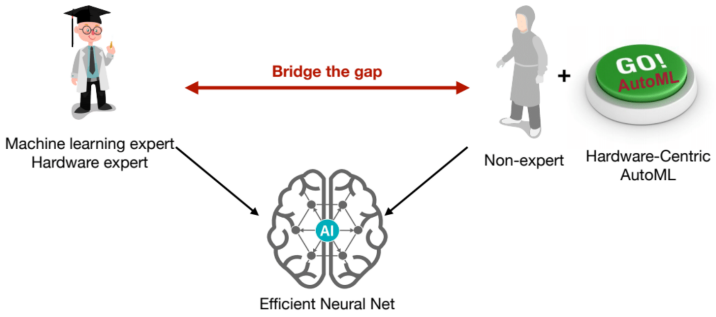
sensitivity analysis는 각 layer들끼리 독립적이라고 가정한 채로 pruning ratio를 계산하므로 최적(optimal)이 아닐 수 있음. 실제로는 하나의 layer만이 아니라 전체 model에 대해 고려해야 하므로 이는 heuristic한 방식임. AMC, NetAdapt 등을 사용하여 이를 개선할 수 있음.

1.4.3. AMC

1. Automatic Pruning

Automatic Pruning은 사람 대신 model이나 engine이 자동으로 pruning ratio를 찾는 방식임.

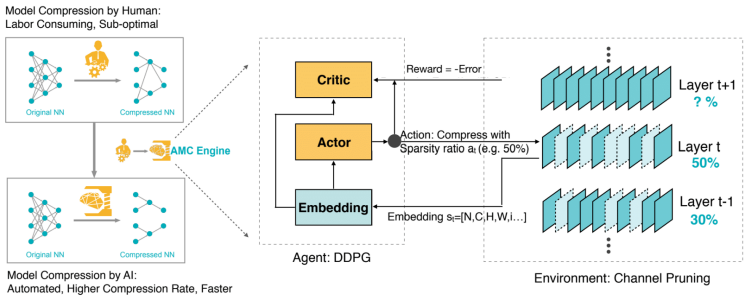
sensitivity analysis의 한계를 넘어서서 model 전체에 대한 pruning ratio를 구하려고 할 때, 고전적인 방식은 수많은 전문가들이 시행착오를 거쳐 최적에 가까운 값을 찾는 것이었음. 강화학습을 활용한 AMC 등을 사용하면 이러한 과정 없이 자동으로 pruning ratio를 얻을 수 있음.



2. AMC

AMC(AutoML for Model Compression)은 model 전반을 고려한 각 layer별 pruning ratio를 구하는 강화학습 모델임. 이는 사람이 직접 적정한 pruning ratio를 찾는 것보다 빠르고, 정확하고(더 작게 pruning함.), 효율적임.

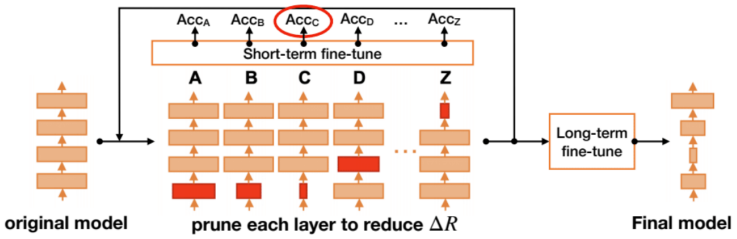
AMC는 학습된 model(CNN)에 대해 동작함. 각 layer에 대한 state를 embedding으로 받고, layer별 pruning ratio(sparsity 지정)를 계산하여 전달한 뒤 전체 model의 accuracy를 계산함. 이후 error를 사용하여 보상을 계산함. 이 과정을 여러 번 반복함.



1.4.4. NetAdapt

NetAdapt는 여러 pruning ratio에 대한 accuarcy를 계산하는 과정을 반복해, 전체 model의 요구사항 (ex. latency, energy 등)을 만족시키면서 각 layer별 최적의 pruning ratio를 찾는 방식임.

아래 그림과 같이, 요구사항을 만족시키면서 각 layer별로 pruning을 적용한 여러 버전에 대해 fine-tuning을 한 뒤 accuracy를 계산함. 여러 버전들 중 가장 높은 accuracy를 가지는 것이 선택됨. 이 과정을 반복하여 최적의 pruning ratio를 적용한 model을 얻을 수 있음. 이때 각 버전에 대해 short term으로 fine-tuning하는 이유는, 중간에 수행하는 학습을 크게 하면 연산량이 너무 많기 때문임.



1.5. Fine-Tuning

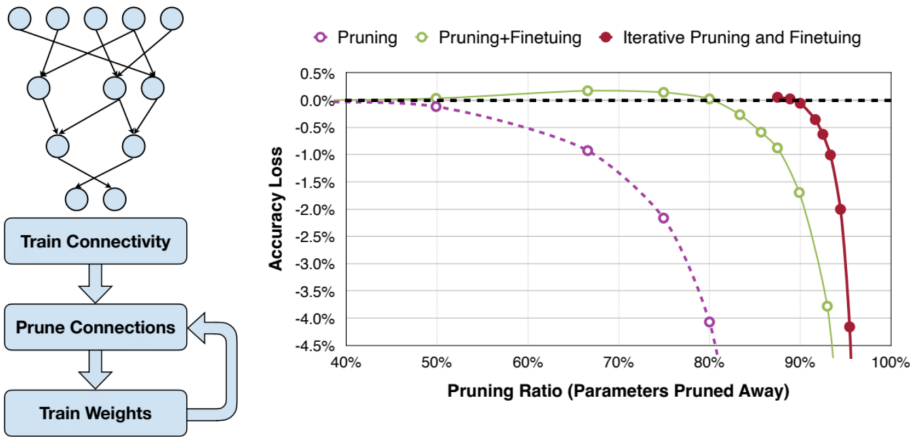
1.5.1. Fine-Tuning

1. Fine-tuning

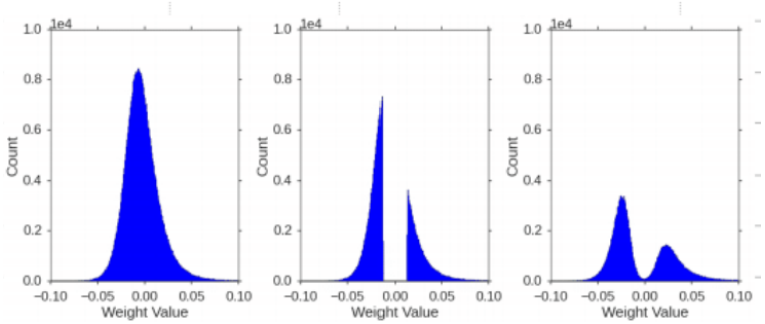
Fine-tuning(파인 튜닝)은 이미 학습된 model을 특정 용도나 데이터셋에 적합하도록 다시 학습시키는 기법임.

참고로, fine-tuning 시에 사용하는 learning rate는 주로 기존 learning rate의 1/100 또는 1/10임.

pruning ratio가 높아질수록 accuracy가 떨어짐. 하지만 pruning 이후 fine-tuning을 적용하면 성능이 개선 또는 유지되고, pruning과 fine-tuning을 반복 수행하면 그 정도를 더 개선할 수 있음. 모델에 따라 다르지만 아래의 그림에서는 성능을 유지하면서 90%정도를 prune할 수 있음.



pruning을 적용하면 0에 가까운(영향이 적은) 가중치들이 삭제되고, fine-tuning을 적용하면 추가적인 학습에 의해 깎인 부분이 매끄러워짐.



2. Regularization
fine-tuning 시에는 regularization을 적절히 적용하는 것이 좋음.

주로 L1, L2 regularization을 사용하는데, 상황에 따라 둘의 성능이 다르기 때문에 실험적으로 적용해 보고 더 나은 방식을 사용해야 함.

1.6. Sparsity를 고려한 System/Hardware

sparsity를 고려하여 설계된 system, hardware를 살펴보자. 이런 식으로 최적화된 architecture를 설계하고 사용할 수 있음.

1.6.1. EIE

1. EIE
EIE(Efficient Inference Engine)는 sparse/compressed DNN model을 위한 accelerator임. EIE는 weight/activation에 대한 fine-grained sparsity를 고려함.

EIE는 weight/activation sparsity를 반영하여 computation과 memory 사용이 적음.

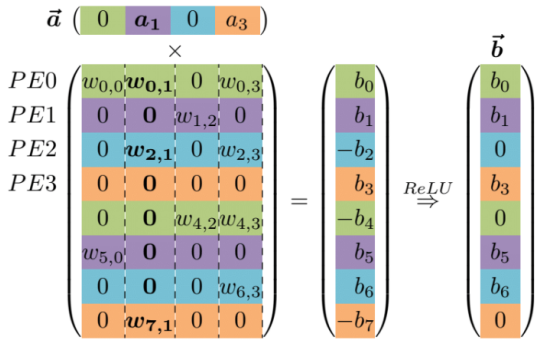
반면에 연산 중간에 사용되는 포인터, index 등에 의한 control flow overhead가 존재하고, FC만을 처리할 수 있음(물론 최근에는 transformer 위주의 model을 주로 사용하기는 함.). 또한 EIE는 TinyML을 위한 accelerator이기 때문에 모든 데이터를 SRAM에 저장하므로 일반적인 경우에 대한 것은 아님.

2. Weight 저장 방법
EIE에서는 weight의 sparsity를 활용함. weight에서 0이 아닌 값과 그 위치에 대한 metadata만을 저장하여 memory 사용을 줄임. 즉, 아래의 그림에서 위는 logical 측면에서의 weight이고, 아래는 physical 측면에서의 weight임.

EIE에서는 weight를 여러 개의 PE(Processing Element)로 나누어 처리함. 아래의 그림에서는 4개의 PE를 사용했음.

memory에는 동일한 PE에 해당되는 weight끼리 묶어서 저장하는데, 값과 그 위치를 저장함. 위치는 relative index와 column pointer라는 두 가지 값을 사용해 나타냄.

logically

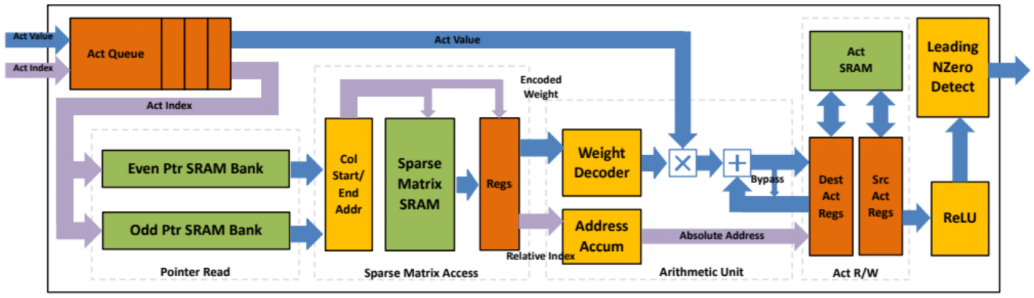


physically

Virtual Weight	$W_{0,0}$	$W_{0,1}$	$W_{4,2}$	$W_{0,3}$	$W_{4,3}$
Relative Index	0	1	2	0	0
Column Pointer	0	1	2	3	5

3. Architecture

EIE의 architecture는 아래와 같음.

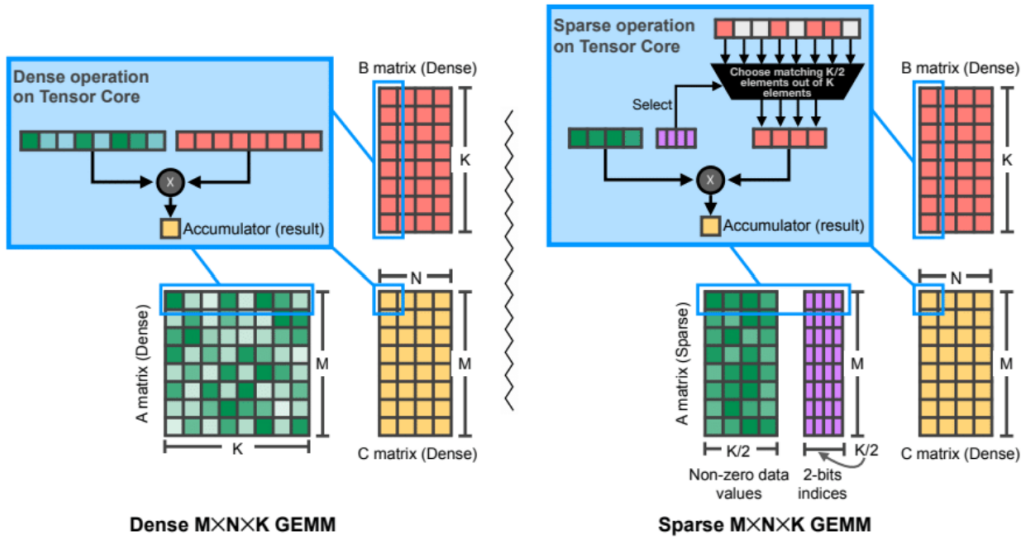


input activation을 받으면 0이 아닌 요소들만 골라서 그 값과 index를 Activation Queue에 넣음. 이후 index를 포인터로 fetch하고 적절한 위치의 W값들을 꺼내 연산함. 이때 W는 quantization에 의해 16 비트를 4비트로 변환한 상태이므로 다시 16비트로 decoding함(W4A16). 이후 ReLU를 거친 뒤 출력을 내보내는데, Nzero Detection Unit에 의해 0인 것들은 제거되고 non zero만 내보냄(activation sparsity 반영).

1.6.2. NVIDA Tensor Core

NVIDA Tensor Core는 DL 연산을 고속으로 처리하는 데에 특화된 processor임. 이는 앞에서 살펴본 N:M sparsity(pattern-based pruning)를 사용하여 weight sparsity를 고려함.

아래와 같이 metadata matrix를 활용해 input activation에서 연산에 사용할 부분만을 추출하고, compress된 matrix와의 연산을 수행함.



1.6.3. TorchSparse/PointAcc

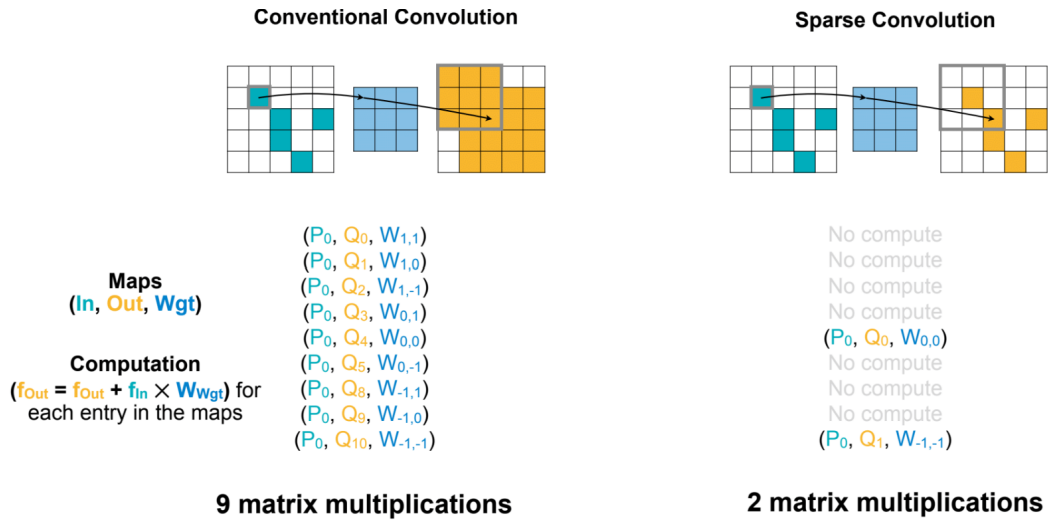
1. TorchSparse

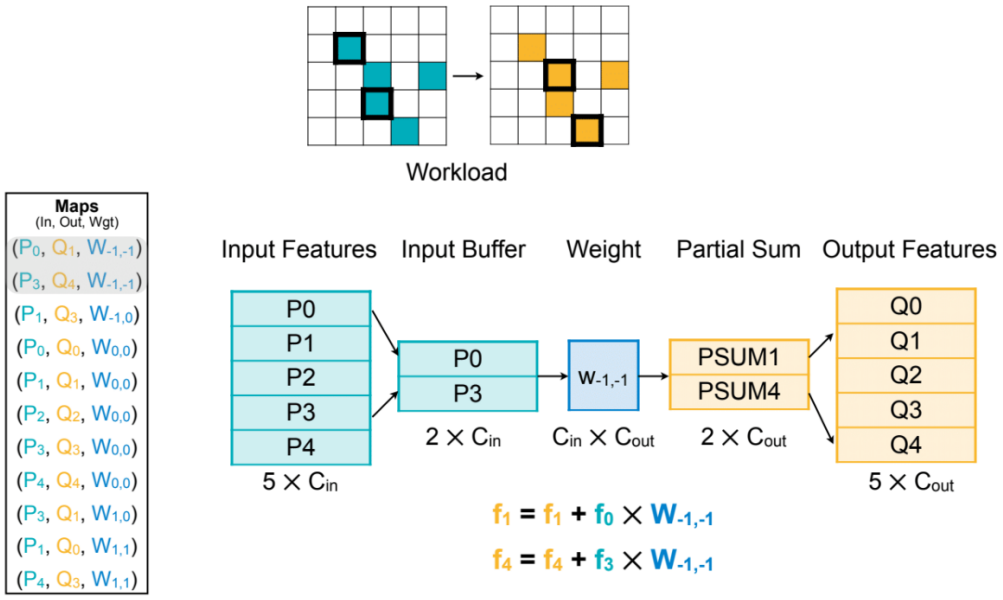
TorchSparse는 sparse convolution 처리를 위한 라이브러리임. activation sparsity를 고려함.

원본 데이터가 sparse하더라도, 단순한 convolution 연산을 반복하면 특정 원소의 값이 여러 부분에 활용되기 때문에 sparsity가 희석됨(dense하게 됨).

TrochSparse의 sparse convolutinon은 출력 tensor가 입력 tensor와 동일한 sparsity pattern을 가지도록 동작함. 아래 그림과 같이 출력이 입력과 동일한 pattern을 가진다고 가정하고, 출력에서 0이 아닌 요소에 대해서만 convolution 연산을 수행함.

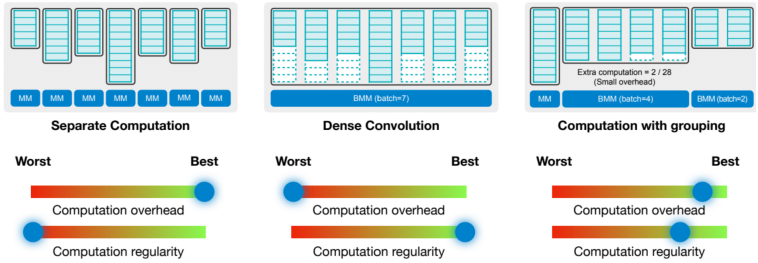
여기에서 입력과 출력의 각 원소는 $P_0, \cdots, P_4$와 $Q_0, \cdots, Q_4$로 인덱싱되고, weight의 인덱스는 -1부터 시작하여 중간이 (0,0)임. sparse convolution에서는 P_i와 weight의 연산으로 Q_j가 도출되는 각 mapping을 $(P_i, Q_j, W_{a,b})$의 형태로 나타냄. 이후 weight의 인덱스에 따라서 mapping 정렬하고, 동일한 weight끼리 묶어서 실질적인 연산을 수행함.





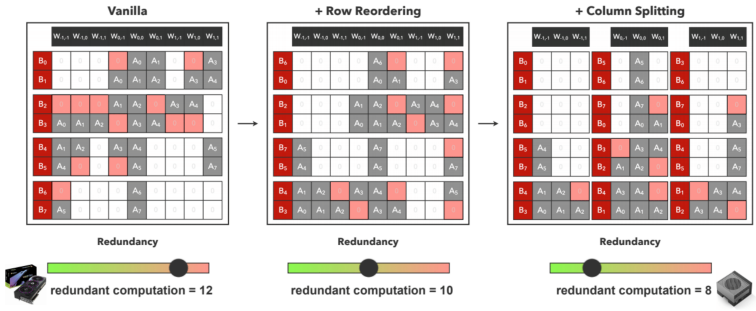
3. Grouping

앞에서 설명한 것과 같이 weight별로 workload를 구성해 연산을 한다면, 각 weight에 대한 mapping의 개수에 따라 computation에 imbalance가 발생함. 만약 weight 각각을 개별적으로 연산한다면 computation regularity가 매우 낮아져 병렬 처리에 불리하고, 모든 weight에 대해 padding을 추가하여 일괄적으로 연산한다면 padding에 따른 computation overhead가 커짐. 이에 따라 TorchSparse에서는 아래의 그림과 같이 mapping의 개수가 유사한 일부 weight들을 grouping하여 이를 해결함.



여기에서 computation과 regularity는 tradeoff 관계에 있다는 것을 알 수 있음.

TorchSparse를 개선한 TorchSparse++에서는 행(입력과 출력 쌍)을 재정렬하고, 이를 활용해 열에 대한 grouping의 효과를 개선하는 등의 기법을 사용함.

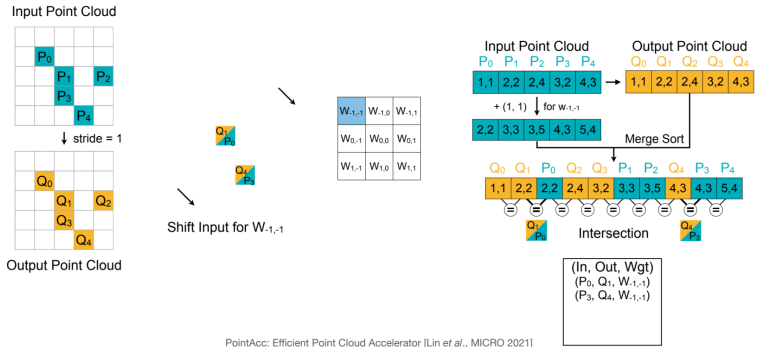


4. PointAcc

PointAcc는 sparse convolution 처리를 위한 하드웨어 accelerator임.

PointAcc는 sparse convolution의 mapping을 효율적으로 찾음. 아래의 그림처럼 input과 output tensor

를 겹쳐놓고, input tensor를 shift하여 output tensor와 일치하는 요소를 mapping으로 추가함. 예를 들어, shift하지 않은 상태에서 일치하는 부분은 $W_{0,0}$에 해당하는 mapping, 오른쪽 아래로 한 칸 shift 한 뒤 일치하는 부분은 $W_{-1,-1}$에 해당하는 mapping임. 이런 식으로 각 weight에 대한 mapping을 전부 찾을 수 있음.



PointAcc: Efficient Point Cloud Accelerator [Lin *et al.*, MICRO 2021]

2. Quantization

2.1. Numeric Data Type

자세한 내용은 '컴퓨터구조(공영호)'를 참고하자.

2.1.1. Integer

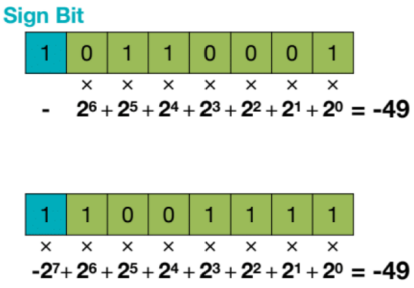
Integer에는 unsigned와 signed 표현이 존재함.

1. Unsigned
음이 아닌 정수를 단순 이진수로 표현함.

2. Signed
sign-magnitude 또는 2의 보수 표현법을 사용함.

Sign-magnitude는 최상위 비트를 부호 비트로 사용하고 나머지 비트로 크기를 나타내는 방식임.

2의 보수 표현법은 양수는 sign-magnitude와 동일하고, 음수는 동일한 크기의 양수에 대해 1의 보수를 취하고(비트를 전부 뒤집고) 1을 더한 것을 값으로 하는 기법임. 비트가 n개인 경우 맨 앞 비트는 -2^{n-1}에 해당하는 것으로 생각할 수도 있음.



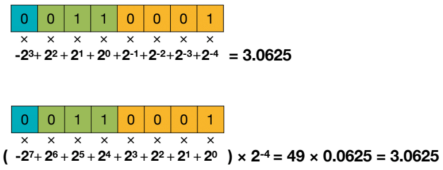
2.1.2. Floating Point

1. Fixed Point
Fixed-point(고정소수점) 방식은 소수점 위치를 고정하여 실수를 표현하는 방식임. 정수부와 소수부의 비트 수를 미리 약속한 후 표현함.

이때 정수부는 n번째 비트의 경우 2의 n-1승에 해당되고, 소수부는 n번째 비트의 경우 2의 -n승에

해당됨. 각 자리의 거듭제곱들을 전부 더하면 해당 소수가 됨.

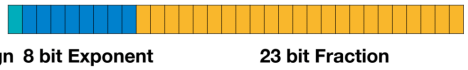
또는 각 비트가 2^0부터 2^{n-1}에 해당되는 것으로 생각하고, 계산 마지막에 소수점 위치에 따른 2의 거듭제곱을 곱해 shift하는 식으로 구현할 수도 있음. 실제로 fixed point의 연산은 정수 연산 이후 2의 거듭제곱만큼의 bit shift를 통해 구현되므로 floating point보다 비용이 적음.



2. Floating Point

Floating-point(부동소수점) 방식은 소수점 위치를 가장 큰 자릿수의 숫자 바로 오른쪽으로 고정하여 표현하는 방식임.

32비트를 사용하는 IEEE 754의 Single Precision(단정밀도) Floating Point(IEEE FP32)의 비트는 Sign bit, Exponent, Mentissa(Fraction)로 구성되고, 나름의 규칙에 따라 저장됨. 또한 exponent 값에 따라 normalized value와 subnormalized value가 존재함. 자세한 내용은 컴퓨터구조 필기를 참고하자.



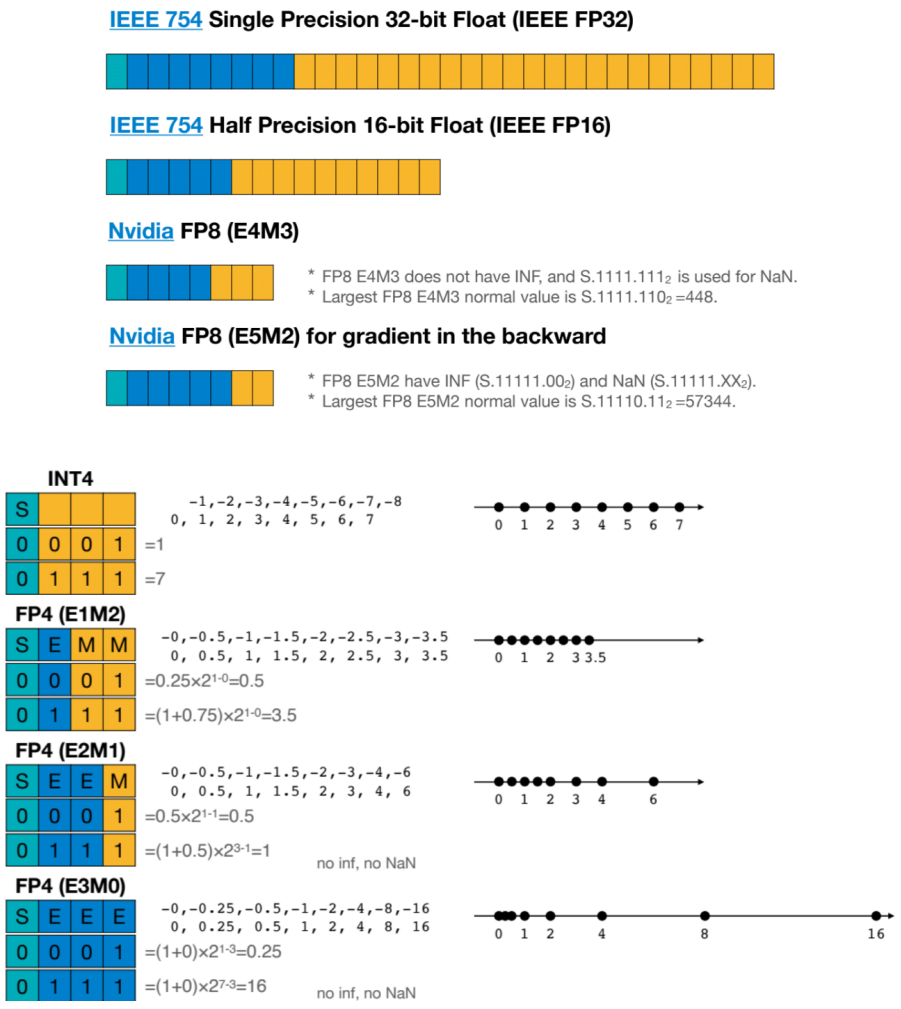
Exponent	Fraction=0	Fraction≠0	Equation
$00_H = 0$	±0	subnormal	$(-1)^{sign} \times \mathbf{Fraction} \times 2^{1-127}$
$01_H \ldots FE_H = 1 \ldots 254$	normal		$(-1)^{sign} \times (1 + \mathbf{Fraction}) \times 2^{\mathbf{Exponent}-127}$
$FF_H = 255$	±INF	NaN	

3. 다양한 Floating Point

더 적은 수의 bit를 사용하여 efficiency를 확보할 수 있음. exponent를 줄이면 range가 줄어들고, fraction을 줄이면 precision이 낮아짐. 즉, 동일한 비트 수에 대해 range와 precision을 tradeoff 관계에 있음.

IEEE FP32 외에도 아래와 같이 floating point를 표현하는 여러 방식들이 존재함. 값을 저장하는 원리는 IEEE FP32와 동일함.

1) IEEE FP16(Half-precision Floating Point, E5M10, IEEE 754)
2) BF16(Brain Float, E4M3, google)
3) FP8(E4M3, nvidia)
3) FP8(E5M2, nvidia)
4) INT4, FP4(E1M2, E2M1, E3M0)



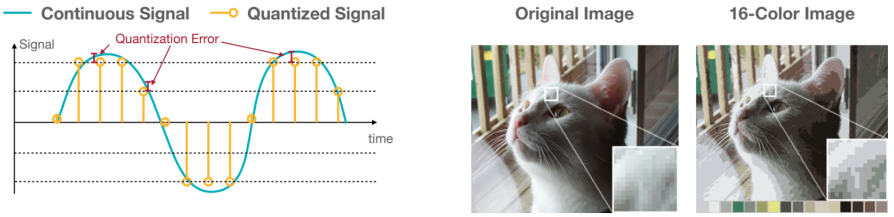
2.2. Quantization

2.2.1. Quantization

1. Quantization

Quantization(양자화)은 연속적(continuous)이거나 많은 양의 값을 포함하는 집합을 이산적인(discrete) 값의 집합으로 변환(제한)하는 것을 말함. 즉, 실수 값을 정수로 변환하는 것 등을 의미함. 이때 n-Bit Quantization은 기존 데이터를 n비트 길이의 integer로 변환하는 것을 의미함.

입력값과 quantization이 적용된 값의 차이를 Quantization Error라고 함. quantization에서는 quantization error를 최소화하는 것이 중요함.

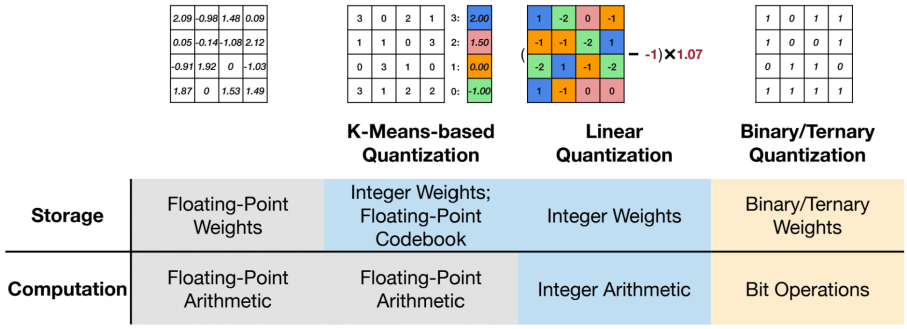


2. Post-training Quantization

학습된 DL model에 quantization을 적용하여 weight 및 activation의 floating point 값을 integer 값 등(INT8이나 FP4 등)으로 변환하고, 이에 따라 관련 연산을 integer로 수행하도록 개선할 수 있음.

quantization을 적용하면 computation의 측면에서는 적은 수의 bit를 처리하는 integer 연산이 더 빠르고 에너지를 덜 먹고, memroy 측면에서는 적은 수의 bit 사용하므로 model size를 줄일 수 있음.

DL model에 대한 quantization으로는 k-means-based quantization, linear quantization, binary quantization 등이 있음. 각 방식들은 memory에 저장하는 방식과 arithmetic 연산 방식의 관점에서 비교할 수 있음. 물론 구현에 따라 각 기법에 대해서 quantization 정도가 달라질 수 있음.

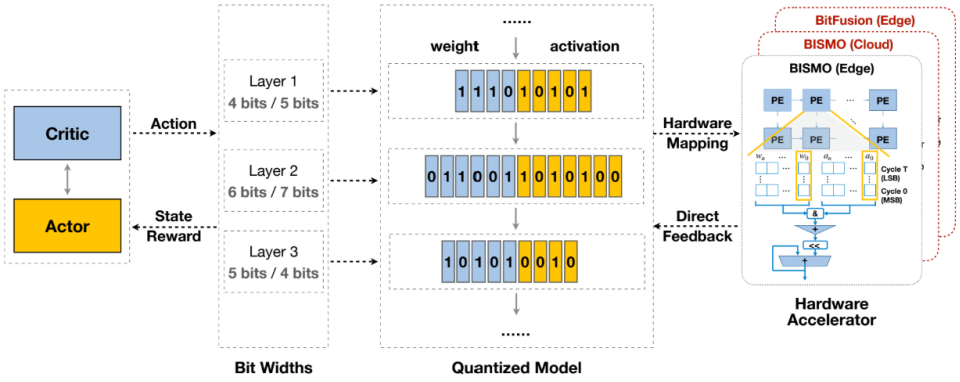


당연하게도 이런 기법은 edge device, 임베디드 시스템 등에서의 model뿐만 아니라, cloud system에서도 중요함.

2.2.2. Mixed-precision Quantization

pruning에서처럼, 각 layer별로 최적화된 quantization(precision)을 적용해 성능을 향상시킬 수 있음.

당연하게도 brute force로는 너무 많은 경우의 수가 존재하므로, 아래와 같이 강화학습을 사용해 최적화할 수 있음.



2.3. K-Means-based Quantization

2.3.1. K-Means-based Quantization

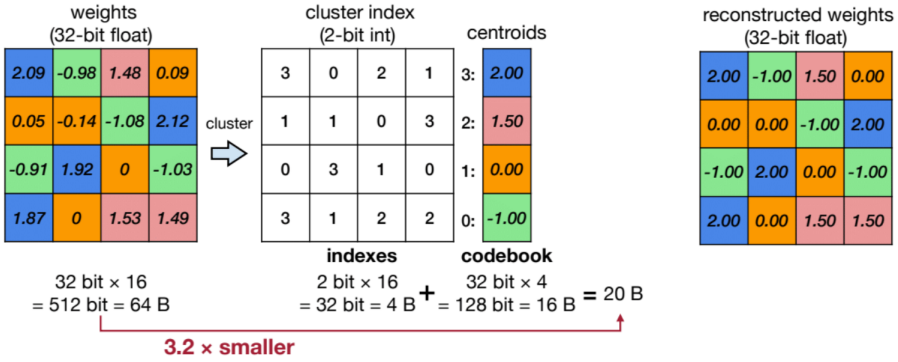
1. K-Means-based Quantization

K-Means-based Quantization은 값을 k개의 group 중 가장 가까운 centroid를 가지는 것으로 clustering 하고, 각 데이터를 해당 group의 값으로 변환하는 기법임. 이때 각 group별 중간값(대푯값)을 Centroid 라고 하고, centroid들을 가지고 있는 자료구조를 Codebook이라고 함.

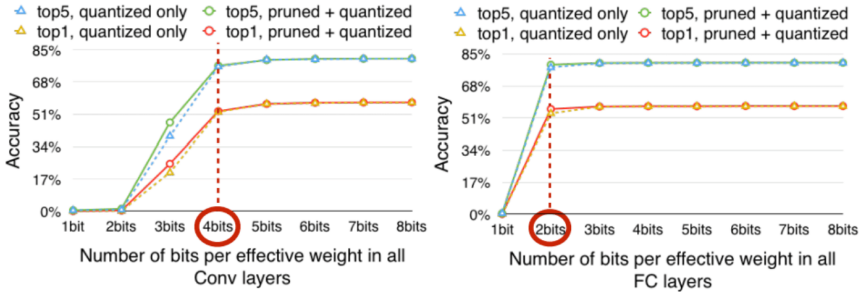
k-means-based quantization에서는 weight를 integer로 저장하고, floating point arithmetic 연산을 수행함. 즉, 해당 값을 연산에 사용할 때는 floating point로 변환함. 이때 codebook의 값을 활용하여 integer를 floating point로 변환하므로 codebook은 floating point임. 이런 방식은 parameter에 대해 높은 precision을 확보하거나 모든 수가 정확할 필요가 없기 때문에 유효함.

FP32를 사용하는 parameter의 개수가 m이라고 하면 model size는 32m bit임. k-means-based로 n-

bit(INTn) quantization을 적용하면 model size는 mn bit이고, codebook은 그 크기가 비교적 작으므로 무시할 수 있음. n이 작을수록 더 많은 memory를 줄일 수 있음.



실험적으로는 convolution layer에 대해서는 4bit, FC layer에 대해서는 2bit로 quantize하는 것이 최적이라고 함.

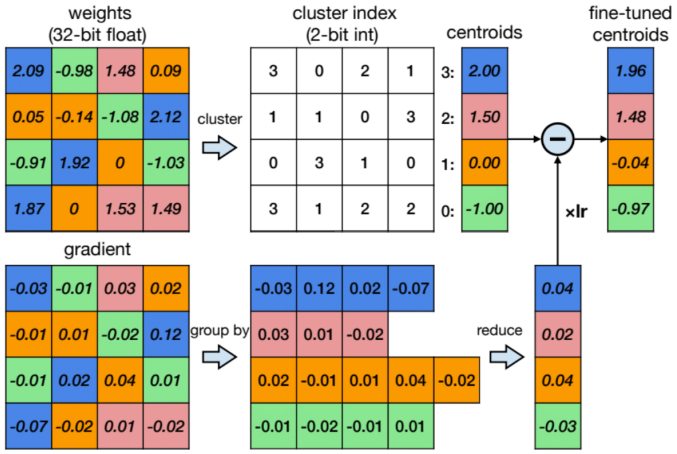


k-means-based quantization은 integer로 저장하면서 memory 측면에서의 이점은 있지만, 이후 dequantize를 수행하여 floating point 연산을 사용하므로 computation에서의 이점은 없음. 하지만 LLM 등 최근의 시스템에서는 memory 측면에서의 제약이 더 많기 때문에 이 방식에도 효용이 존재함.

2. Fine-Tuning

k-means based quantization의 codebook 값은 fine-tuning에 의해 최적화됨.

아래와 같은 과정을 거쳐 backpropagation이 수행됨. 1) 각 weight별 gradient를 계산한 gradient matrix를 구성함. 2) 동일한 group에 해당하는 gradient끼리 모아서 합함. (group별 gradient가 도출됨.) 3) group별 gradient에 learning rate을 곱해서 기존의 codeboook에 더함.



2.3.2. Pruning과 Quantization

1. Huffman Coding

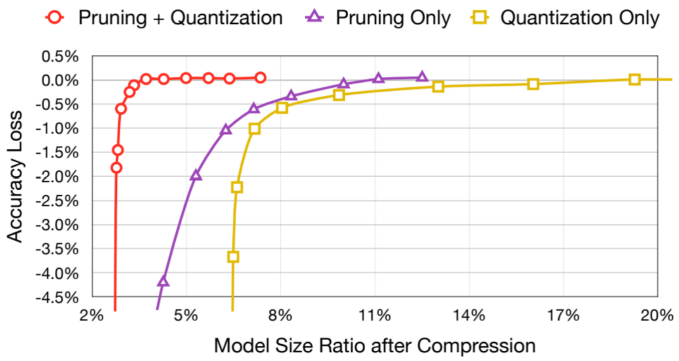
Huffman Coding은 빈도수가 높은 대상은 짧은 code로, 빈도수나 낮은 대상은 긴 code로 압축하는 기법임.

quantize된 model에 적용하면 각 weight 별 개수가 많은 것은 적은 비트를 사용해 저장하고, 개수가 적은 것은 많은 비트를 사용해 저장할 수 있음.

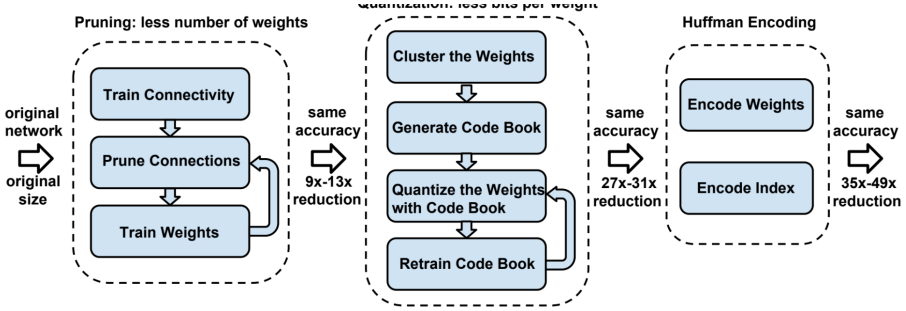
2. Pruning과 Quantization

pruning과 quantization, huffman coding을 모두 적용하여 model compression을 수행할 수 있음. 이 경우 pruning이나 quantization 중 하나만 적용했을 때보다 훨씬 많이 compress할 수 있음.

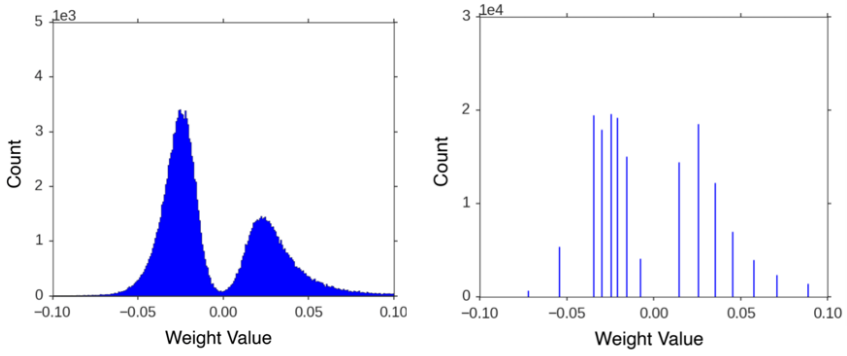
아래의 그래프는 AlexNet에 pruning, quantization을 적용했을 때 compression 정도에 따른 accuarcy를 나타냄. pruning과 quantization을 모두 적용한 것이 가장 많은 compression이 가능함.



pruning을 적용한 뒤 fine-tuning하고, 이후 quantization을 적용한 뒤 fine-tuning하고, 마지막으로 huffman coding을 적용해 더 compress함.



pruning과 fine-tuning을 적용하면 weight에서 0에 가까운 부분이 삭제되고, quantization을 적용하면 특정 값들로 discrete한 분포를 가지게 됨.



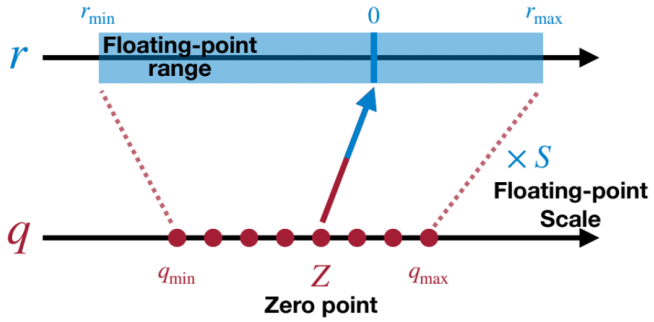
2.4. Linear Quantization : Concept

2.4.1. Linear Quantization

1. Linear Quantization

Linear Quantization은 weight의 저장과 arithmetric 연산 모두 integer로 수행하는 quantization 기법임.

linear quantization은 Zero point(Z)와 Scale(S)이라는 값(Linear Quantization Parameter)을 활용해 weight를 quantize하고, 입력과 연산함. Z는 quantize된 집합에서 0에 해당하는 부분을 지정하는 integer 값이고, S는 scaling을 하는 floating point 값임. Z와 S만 있으면 linear quantization이 가능함.



linear quantization에서 floating point 값을 매핑(quantize)하는 integer 집합(q를 구성하는 값에 대한 집합)은 당연하게도 integer에 사용하는 비트 수에 따라 정의됨. integer 저장에 n비트를 사용한다면 해당 비트로 표현할 수 있는 2의 보수만큼인 $-2^{n-1} \sim 2^{n-1}-1$의 정수를 사용해 floating point를 매핑함. 예를 들어, 2비트를 사용하면 floating point 값들을 $-2 \sim 1$로 매핑함. 대부분의 시스템은 8비트 크기의 저장 공간을 제공하므로 $n = 8$비트가 괜찮은 성능을 보인다고 함(W8A8).

2. 저장 방법

floating point weight를 r, quantize를 적용한 integer weight를 q라고 하자. Z와 S 사이에는 아래의 수식이 성립함. 이 수식을 사용해 quantize된 q로부터 reconstruct할 수 있음.

$$r = (q - Z) \cdot S$$

이 수식을 사용하여 r을 q로 quantize하는 것도 당연하게 계산할 수 있음. 이때 r은 integer이므로 가까운 정수로 round(반올림)함.

$$q = round\,(\frac{1}{S} \cdot r + Z)$$

이때 Z와 S는 r과 q의 최댓값과 최솟값에 의해 결정됨. 아래와 같이 $r_{max} = S(q_{max} - Z)$와 $r_{min} = S(q_{min} - Z)$을 연립하여 S를 계산할 수 있고, S을 알았으면 Z 또한 계산할 수 있음. 이때 Z는 integer이므로 가까운 정수로 round함. 또한 대상이 정규분포이면(아래에서 설명함.) Z가 0이고 $|r_{max}| = |r_{min}|$이어서 S를 더 단순하게 계산할 수 있음.

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} = \frac{r_{min}}{q_{min} - Z} = \frac{-|r_{max}|}{q_{min}} = \frac{|r_{max}|}{2^{N-1}}$$

$$Z = round\,(q_{min} - \frac{r_{min}}{S})$$

3. 연산 방법

1) FC layer에서의 연산

linear quantization을 적용한 FC layer의 weight에서 연산($Y = WX + b$)은 아래와 같이 정리할 수 있음. 이때 W와 b는 일반적으로 평균이 0인 정규분포와 유사한 형태를 가짐. 즉, 값의 분포가 이미 0에

맞춰져 있으므로 $Z_W = Z_b = 0$이라고 할 수 있음(이제 W와 b에 대해서 Z는 0인 것으로 가정함.). 또한 $S_W S_X$와 S_b는 각각 값을 scaling하는데, b는 $S_W S_X$로 scaling된 값에 더해져야 하므로 $S_b = S_W S_X$여야 함. W와 b, 그리고 S, Z는 연산 전에 계산되어 있으므로 $q_b - Z_X q_w$에 대한 연산 또한 미리 수행될 수 있고, 이를 q_{bias}로 표기함.

$$\mathbf{Y} = \mathbf{WX} + \mathbf{b}$$

$$S_{\mathbf{Y}}\left(\mathbf{q}_{\mathbf{Y}} - Z_{\mathbf{Y}}\right) = S_{\mathbf{W}}\left(\mathbf{q}_{\mathbf{W}} - Z_{\mathbf{W}}\right) \cdot S_{\mathbf{X}}\left(\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\right) + S_{\mathbf{b}}\left(\mathbf{q}_{\mathbf{b}} - Z_{\mathbf{b}}\right)$$

$$\downarrow Z_{\mathbf{W}} = 0$$

$$S_{\mathbf{Y}}\left(\mathbf{q}_{\mathbf{Y}} - Z_{\mathbf{Y}}\right) = S_{\mathbf{W}}S_{\mathbf{X}}\left(\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}}\right) + S_{\mathbf{b}}\left(\mathbf{q}_{\mathbf{b}} - Z_{\mathbf{b}}\right)$$

$$\downarrow Z_{\mathbf{b}} = 0, \quad S_{\mathbf{b}} = S_{\mathbf{W}}S_{\mathbf{X}}$$

$$S_{\mathbf{Y}}\left(\mathbf{q}_{\mathbf{Y}} - Z_{\mathbf{Y}}\right) = S_{\mathbf{W}}S_{\mathbf{X}}\left(\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}} + \mathbf{q}_{\mathbf{b}}\right)$$

$$\downarrow$$

$$\mathbf{q}_{\mathbf{Y}} = \frac{S_{\mathbf{W}}S_{\mathbf{X}}}{S_{\mathbf{Y}}}\left(\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} + \underbrace{\mathbf{q}_{\mathbf{b}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}}}_{\text{Precompute}}\right) + Z_{\mathbf{Y}}$$

$$\downarrow q_{bias} = \mathbf{q}_{\mathbf{b}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}}$$

We will discuss how to compute activation zero point in the next lecture.

$$\mathbf{q}_{\mathbf{Y}} = \frac{S_{\mathbf{W}}S_{\mathbf{X}}}{S_{\mathbf{Y}}}\left(\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} + \mathbf{q}_{bias}\right) + Z_{\mathbf{Y}}$$

정리하면 아래의 수식과 같은 연산이 수행됨.

$$\mathbf{Y} = \mathbf{WX} + \mathbf{b}$$

$$\downarrow Z_{\mathbf{W}} = 0, \quad Z_{\mathbf{b}} = 0, \quad S_{\mathbf{b}} = S_{\mathbf{W}}S_{\mathbf{X}}, \quad \mathbf{q}_{bias} = \mathbf{q}_{\mathbf{b}} - Z_{\mathbf{X}}\mathbf{q}_{\mathbf{W}}$$

$$\mathbf{q}_{\mathbf{Y}} = \underbrace{\frac{S_{\mathbf{W}}S_{\mathbf{X}}}{S_{\mathbf{Y}}}}_{\text{Rescale to } N\text{-bit Int}}\left(\underbrace{\mathbf{q}_{\mathbf{W}}\mathbf{q}_{\mathbf{X}} + \mathbf{q}_{bias}}_{N\text{-bit Int Mult. 32-bit Int Add.}}\right) + \underbrace{Z_{\mathbf{Y}}}_{N\text{-bit Int Add}}$$

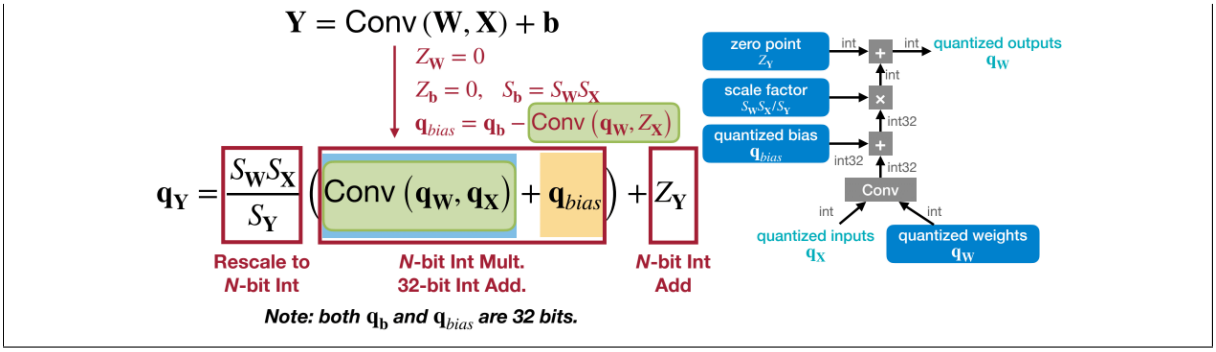
Note: both $\mathbf{q}_{\mathbf{b}}$ and $\mathbf{q}_{bias}$ are 32 bits.

이때 경험적으로 $\frac{S_W S_X}{S_Y}$는 항상 (0,1)에 속하는데, 이를 $\frac{S_W S_X}{S_Y} = 2^{-n} M_0$로 표현할 수 있음. M_0는 $[0.5, 1)$에 속하는 fixed point로, 실수가 항상 2의 거듭제곱으로 표현되지는 않기 때문에 사용하는 보정값임. 즉, $\frac{S_W S_X}{S_Y}$를 곱하는 것은 fixed point 연산(실제로는 정수 연산 이후 bit shift가 적용됨.)과 bit shift로 수행될 수 있음.

quantize된 값 끼리의 곱셈 결과는 기존 N비트에 대해 overflow가 발생할 수 있으므로 직후의 덧셈 연산은 32비트로 수행함.

즉, 입력과 출력, weight 모두 integer로 quantize된 상태로 처리하는데, 내부적인 연산 또한 integer로 수행됨. 또한 파란색 부분에서 대부분의 연산이 발생함. k-means-based quantization에서는 32비트 floating point 연산을 했는데, 여기에선 32비트 integer 연산을 사용하므로 연산 측면에서의 이점이 존재함.

2) Convolution Layer에서의 연산
입력이 filter의 각 weight와 곱해진다는 것을 제외하면 convolution layer에서의 연산도 fc layer와 동일함.



linear quantization을 사용하면서 weight-only quantization을 적용하기도 함. 예를 들어, weight는 INT4로 quantize하고, activation은 FP16으로 연산하는 W4A16(Weight 4 Activation 16)이 있음.

2.5. Linear Quantization : Post-Training Quantization

post-training quantization에서 model에 대한 linear quantizaton을 어떻게 잘 적용할 수 있을지 알아보자. 즉, 어떤 것을 고려하여 weight와 activation에 대해 최적의 S, Z를 얻을 수 있는지 알아보자.

2.5.1. Quantization Granularity

Quantization Granularity는 S와 Z를 계산하고 적용하는 단위에 대한 granularity로, per-tensor, per-channel, group 등의 방식이 있음. 이를 고려하여 weight를 quantization할 수 있음.

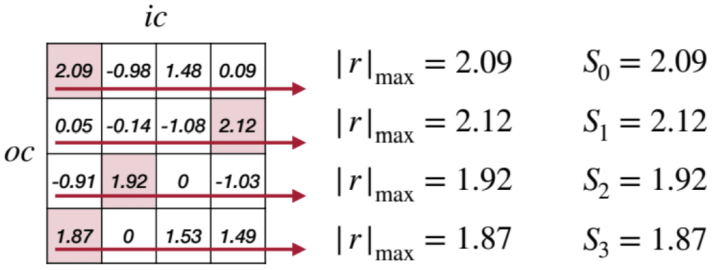
물론 weight가 정규분포인 경우 Z는 0이므로 S만을 구한다고 생각할 수 있음.

1. Per-Tensor Quantization
Per-Tensor Quantization은 전체 tensor에 대해 하나의 S와 Z를 계산하고 적용하는 방식임.

이는 대상이 많은 경우에는 잘 동작하지만, 대상이 적은(model size가 작은) 경우에는 accuracy 감소가 심함. 이는 각 대상(channel)마다 상이한 dynamic range를 가질 수 있는데 그 개수가 적으면 각 channel의 영향이 커지기 때문임. S와 Z를 per-channel로 계산하는 것으로 개선할 수 있음.

2. Per-Channel Quantization
Per-Channel Quantization은 각 channel별로 하나의 S와 Z를 계산하고 적용하는 방식임.

더 작은 단위로 계산했으므로 당연하게도 per-tensor quantization에 비해 reconstruction error가 더 작아지지만, 다만 더 많은 scaling factor를 저장해야 하므로 추가적인 memory를 사용하게 됨.



3. Group Quantization
Group Quantization은 여러 channel이나 weight의 일부를 group으로 묶어서 각 group에 대한 S와 Z를 계산하고 공유 적용하는 방식임. per-tensor와 per-channel에 대한 중간 수준의 quantization이라고 생각할 수 있음.

group quantization에서 각 channel 등은 나름의 parameter(linear quantization parameter)를 가지고, group의 대한 parameter 또한 가짐. 이때 각 요소 별로 사용하는 비트 수는 해당 요소가 사용하는 비트 수와 group에 대한 비트 수를 더해서 계산할 수 있는데, 이와 같이 한 요소를 표현하는 데에 실질적으로 사용하는 비트 수를 Effective Bit Width라고 함.

effective bit width의 관점에서 볼 때, group quantization은 linear quantization parameter에 대해 더 작은 공간을 사용하면서 accuracy를 보장함. 특히 최근 nvidia의 blackwell 등에서는 FP4에 대한 고속 연산을 수행하는 tensor core를 제공하는데, 데이터를 4비트 수준까지 quantization하려면 데이터 분포에 따라 세밀한 처리가 필요함. group quantization을 사용하면 memory를 과하게 사용하지 않으면서 더 flexible하게 accuracy를 확보할 수 있음.

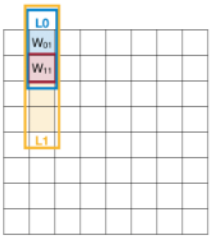
group 구조에 따라 여러 방식이 존재하는데, 대표적으로는 VSQ와 MX가 있음. 동일한 규칙으로 multi-level scaling을 이해할 수 있음.

1) VSQ(VS-Quant, Per-Vector Scaled Qunatization)
VSQ는 전체 tensor에 대한 scaling factor γ와, tensor 내부 벡터에 대한 scaling factor S_q를 각각 계산하여 곱하는 방식임. 즉, 2-level scaling임.

γ는 tensor에 대해 coarse grain으로 계산한 floating point scaling factor임. floating point이므로 비용이 크고 tensor당 하나만 존재함. S_q는 각 벡터에 대해 fine grain으로 계산한 integer scaling factor임. integer이므로 비용이 적고 각 벡터 별로 존재함.

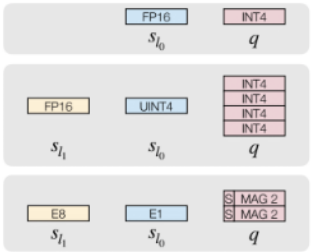
참고로 fine grain이 granularity가 더 작다고 한다. 의미적인 정리를 하자.

2) MX4, MX6, MX9
MX4, MX6, MX9에서는 아래 그림과 같은 구조를 활용하여 group quantization을 적용함. 이때 MX 뒤에 붙는 숫자는 effective bit width를 말함.



$$r = (q - z) \cdot s_{l_0} \cdot s_{l_1} \cdot \cdots$$

r : real number value
q : quantized value
z : zero point ($z = 0$ is symmetric quantization)
s : scale factors of different levels



Quantization Approach	Data Type	L0 Group Size	L0 Scale Data Type	L1 Group Size	L1 Scale Data Type	Effective Bit Width
Per-Channel Quant	INT4	Per Channel	FP16	-	-	4
VSQ	INT4	16	UINT4	Per Channel	FP16	4+4/16=4.25
MX4	S1M2	2	E1M0	16	E8M0	3+1/2+8/16=4
MX6	S1M4	2	E1M0	16	E8M0	5+1/2+8/16=6
MX9	S1M7	2	E1M0	16	E8M0	8+1/2+8/16=9

2.5.2. Dynamic Range Clipping

Dynamic Range는 데이터나 신호가 표현할 수 있는 최소값과 최대값 사이의 범위임. 여기에서 dynamic range는 r에 대한 range를 의미하고, Dynamic Range Clipping은 실제 dynamic range에서의 clipping을 통해 r_{min}과 r_{max}를 지정하는 것을 말함. Clipping은 값의 범위를 제한하는 것임.

학습 이후 weight는 dynamic range가 고정되는 반면에, activation은 각 입력에 따라 다른 dynamic range를 가지게 됨. 이에 따라 activation에 model을 deploy(배포)하기 전에 activation에 대한 통계적인 정보를 모아 dynamic range를 정적으로 추정하고, linear quantization parameter를 계산하여 quantization을 적용할 수 있음. 또한 여기서는 다루지 않지만 dynamic range를 infernece 도중에 동적으로 업데이트하는 방식도 존재한다고 함.

동적으로 dynamic range를 계산할 때 주로 사용하는 방법으로는 아래와 같은 2가지 기법이 있음.

1. EMA
EMA(Exponential Moving Average)를 학습 중에 반복 계산하여 dynamic range를 추정할 수 있음.

아래와 같이 특정 batch에 대한 dynamic range 추정치는 이전 batch에서의 추정치와 해당 시점에서의 실제 값을 모두 활용하여 계산함. 이때 α로는 이전 batch의 dynamic range를 얼마나 반영할지를 적절히 지정함.

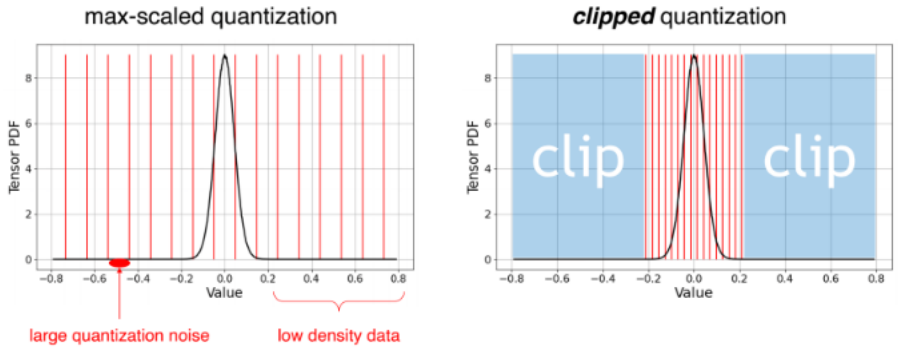
$$\hat{r}^t_{\max,\min} = \alpha \cdot r^t_{\max,\min} + \hat{r}^{t-1}_{\max,\min}$$

이를 통해 여러 batch에 대해 평균적인 dynamic range 추정치를 얻을 수 있음. 하지만 이건 학습 과정에 관여가 가능한 경우에만 가능함. 즉, 직접 학습하는 경우에만 가능하고 다른 곳에서 학습된 모델을 quantize하는 일반적인 경우에는 적용이 불가능함.

2. Using Calibration Batch

Calibration Batch는 dynamic range 추정을 위해 학습된 model에 넣는 데이터임. calibration batch를 model에 입력하고, 각 layer에서의 activation을 얻어 dynamic range 추정치를 계산할 수 있음.

activation이 정규분포 형태일 때, 양쪽 극단에 위치하는 적은 수의 값들까지 dynamic range가 포함하게 되면 아래의 그림과 같이 quantization과 관련된 표현력이 낮아질 수 있음. 이에 따라 적정한 dynamic range를 계산하고 활용하는 것이 중요함.



최적의 dynamic range를 찾는 방법으로는 아래와 같은 것들이 있음. 여기에서는 정규분포 형태임을 가정하므로, $|r_{max}|$만 구하면 됨.

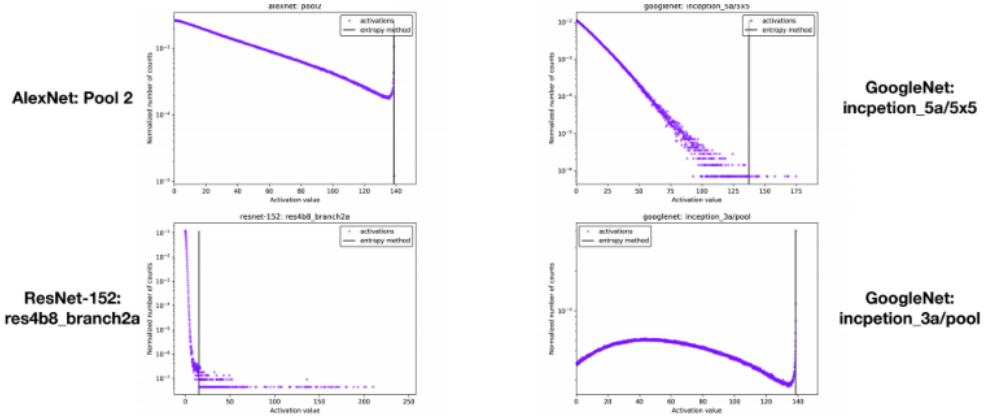
1) 특정 input과, 해당 input을 quantize하고 reconstruct한 값 사이의 MSE가 최소가 되도록 하는 dynamic range($|r_{max}|$)를 찾음. 특히 값이 라플라스 분포 $(0, b)$를 따르는 경우 2, 3, 4-bt quantization 일 때 $|r_{max}|$가 각각 2.83b, 3.89b, 5.03b이면 최적이라고 함. 하지만 activation이 대체로 이런 분포를 따르지는 않음.

2) KL divergence를 최소화하는 dynamic range($|r_{max}|$)를 찾음. KL Divergence는 두 확률분포 사이의 차이(quantization에 의한 데이터 손실)를 나타내는 값으로, 아래와 같은 수식으로 계산됨.

$$D_{KL}(P||Q) = \sum_{i}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)}$$

참고로 Calibration(눈금 메기기)은 교정 및 조정 작업을 의미함.

KL divergence를 활용하면 아래와 같이 dynamic range를 찾을 수 있음.



2.5.3. Rounding

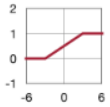
앞에서 다룬 것처럼 linear quantization에서는 rounding이 수행됨. 이때 Rounding은 반올림 뿐만 아니라 지정한 수의 집합으로 변환하는 것을 의미함.

반올림(가장 가까운 값으로의 rounding)이 직관적으로는 그럴듯해 보이지만, 실제로는 최적이 아닐 수 있음. 특히 weight의 각 값이 실제로는 거의 거의 유사한데 반올림에 의해 아예 다른 값이 된다면 오차가 커짐. 실제로는 reconstruction error를 최소화하는 방법이 최적의 rounding임.

반올림($\lfloor W \rceil$), 올림($\lceil W \rceil$), 내림($\lfloor W \rfloor$) 등을 상황에 맞게 적용할 수 있고, $\lfloor \lfloor W \rfloor + \delta \rceil$와 같은 수식을 적용할 수도 있음. δ는 [0,1]에 속하면서 학습 가능한 값으로, input과 shape이 같은 값을 sigmoid 등에 넣은 것임.

$$\text{argmin}_{\mathbf{V}} \|\mathbf{W}\mathbf{x} - \tilde{\mathbf{W}}\mathbf{x}\|_F^2 + \lambda f_{reg}(\mathbf{V})$$
$$\rightarrow \text{argmin}_{\mathbf{V}} \|\mathbf{W}\mathbf{x} - [\lfloor \mathbf{W} \rfloor + \mathbf{h}(\mathbf{V})]\mathbf{x}\|_F^2 + \lambda f_{reg}(\mathbf{V})$$

$\mathbf{x}$ is the input to the layer, $\mathbf{V}$ is a random variable of the same shape
$\mathbf{h}()$ is a function to map the range to $(0,1)$, such as rectified sigmoid
$f_{reg}(\mathbf{V})$ is a regularization that encourages $\mathbf{h}(\mathbf{V})$ to be binary



2.6. Binary/Ternary Quantization

더 적은 비트 수를 사용하는 quantization을 알아보자. 물론 비트 수가 적어진다고 최적은 아니고, 최근에는 4비트로 quantize하는 것의 성능이 가장 뛰어나다고 함.

2.6.1. Binary Quantization

1. Binarization

Binarization(이진화)는 weight나 activation 각각을 2가지 값 중 하나로 변환하는 기법임. binarization 방법으로는 아래와 같은 것들이 있음. 값을 이진화하면 1개의 비트로 표현이 가능해짐.

1) Deterministic Binarization(결정적 이진화)
하나의 threshold를 지정하고 각 값에 대해 threshold를 넘으면 1, 넘지 못하면 -1로 변환하는 방법. 이때 threshold는 주로 0으로 하고, 이 경우 sign function(부호 반환)과 동일하게 동작함.

직관적이고 쉬운 방법이지만 accuracy degradation이 심함.

2) Stochastic Binarization(확률적 이진화)
입력 데이터 또는 특정 통계에 따른 확률에 의해 1 또는 -1로 변환하는 방법. 예를 들어, Binary Connect(BC)에서는 아래와 같이 확률을 계산하여 변환함.

$$q = \begin{cases} +1, & \text{with probability } p = \sigma(r) \\ -1, & \text{with probability } 1-p \end{cases}, \quad \text{where } \sigma(r) = \min(\max(\frac{r+1}{2}, 0), 1)$$

확률적 처리에 따라 무작위 값을 생성해야 하므로 구현이 까다로움.

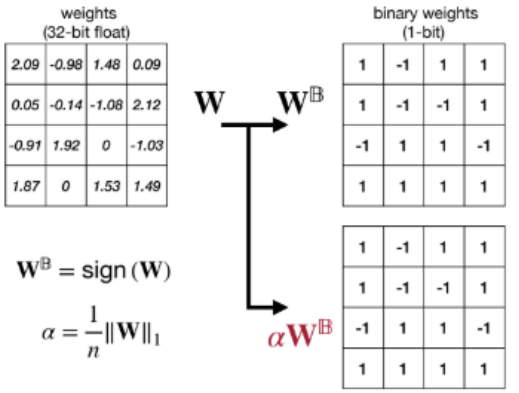
2. BWN

BWN(Binary Weight Network)는 weight만 binarization하고, activation은 binarization하지 않은 NN임.

weight에 sign function을 적용하여 1비트 이진 값으로 저장하고, 아래와 같이 원본 값의 절댓값에 대해 평균을 계산하여 scaling factor로 사용함. 이후 forward propagation에서 저장된 값과 scaling factor를 곱해 활용함.

$$\alpha = \frac{1}{n}||W||_1$$

단순히 binarization만을 적용하고 scaling factor는 사용하지 않는 방식에 비해 accuracy를 확보할 수 있음.



AlexNet-based Network	ImageNet Top-1 Accuracy Delta
BinaryConnect	-21.2%
Binary Weight Network (BWN)	0.2%

3. BNN/XNOR-Net

BNN(Binary Neural Network)는 weight과 activation 모두 binarization는 NN임. 특히, XNOR-Net은 XNOR을 활용해 연산을 수행하는 BNN임.

XNOR는 두 수가 같으면 1, 다르면 0을 반환하는 연산으로, 하드웨어적으로 구현하기 쉬움. XNOR로 binarization된 weight와 입력 사이의 연산을 효율적으로 수행할 수 있음. binarization된 값을 논리적으로는 -1과 1로 생각할 수 있지만, 실제로는 0과 1로 저장됨. 이에 따라 실제 값을 아래와 같이 XNOR 연산을 하면 논리적 값의 연산과 같은 결과를 얻을 수 있음. 아래의 연산은 단순히 W와 입력의 각 원소에 대해 XNOR 연산을 수행하고, 그 결과에 2를 곱한(shift) 뒤 원소의 개수만큼을 뺀 것임.

여기서 popcount는 여러 값들 중 1의 개수를 세는 연산으로, XNOR와 같이 하드웨어적으로 구현이 쉬움. 여기에서 popcount를 사용하면 0과 1의 덧셈 연산을 한 번에 수행할 수 있음.

$$y_i = -n + 2 \cdot \sum_j W_{ij} \; xnor \; x_j = -n + 2 \cdot popcount(W_i \; xnor \; x_j)$$

이렇게 weight와 activation 모두에 대해 binarization을 적용하고 XNOR, popcount를 활용하면 memory와 computation에서의 성능을 향상시킬 수 있음. 하지만 XNOR-Net까지 개선해도 무시할 수 없는 정도의 accuracy degradation가 발생함. 이에 따라 ternary quantization을 적용할 수 있음.

input	weight	operations	memory	computation
$\mathbb{R}$	$\mathbb{R}$	+ ×	1×	1×
$\mathbb{R}$	$\mathbb{B}$	+ -	~32× less	~2× less
$\mathbb{B}$	$\mathbb{B}$	xnor, popcount	~32× less	~58× less

2.6.2. Ternary Quantization

1. TWN

TWN(Ternary Weight Network)는 weight를 3가지 값(1, 0, -1) 중 하나로 변환하는 기법임.

아래의 수식과 같이 각 값(r)과 threshold(Δ)를 비교하여 r_t, 0, $-r_t$로 quantize됨.

$$q = \begin{cases} r_t & (r > \Delta) \\ 0 & (|r| \le \Delta) \\ -r_t & (r < -\Delta) \end{cases}$$

Δ는 아래와 같이 계산할 수 있음. 즉, weight의 각 값의 절댓값의 평균에 0.7을 곱한 것임. 이때 0.7은 실험적으로 구해진 값임.

$$\Delta = 0.7 \times \mathbb{E}(|r|)$$

이때 r_t는 연산 시에 사용하는 scaling factor로, 아래와 같이 0이 아닌 값으로 quantize된 원본 값들의 절댓값의 평균임. 실제로 weight는 1, 0, -1 중 하나를 값으로 가지므로 2비트로 저장됨.

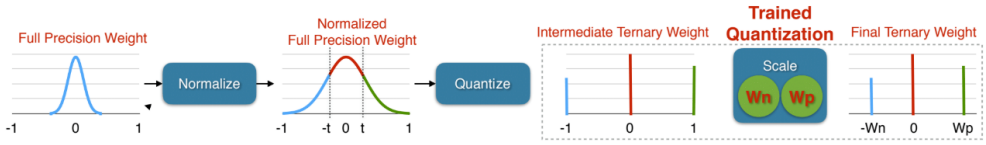
$$r_t = \mathbb{E}_{|r|>\Delta}(|r|)$$

binary quantization에 비해 개선되긴 하지만, 그래도 충분한 accuracy가 확보되지 않을 수 있음.

2. TTQ
TTQ(Trained Ternary Quantization)는 TWN과 동일한데, 아래와 같이 r_t(heuristic함.) 대신에 학습 가능한 parameter w_p와 $-w_n$을 사용하는 방식임.

$$q = \begin{cases} w_p & (r > \Delta) \\ 0 & (|r| \le \Delta) \\ -w_n & (r < -\Delta) \end{cases}$$

TWN보다 accuracy가 살짝 더 개선됨.



ImageNet Top-1 Accuracy	Full Precision	1 bit (BWN)	2 bit (TWN)	TTQ
ResNet-18	69.6	60.8	65.3	66.6

2.7. Quantization-Aware Training

2.7.1. Quantization-Aware Training

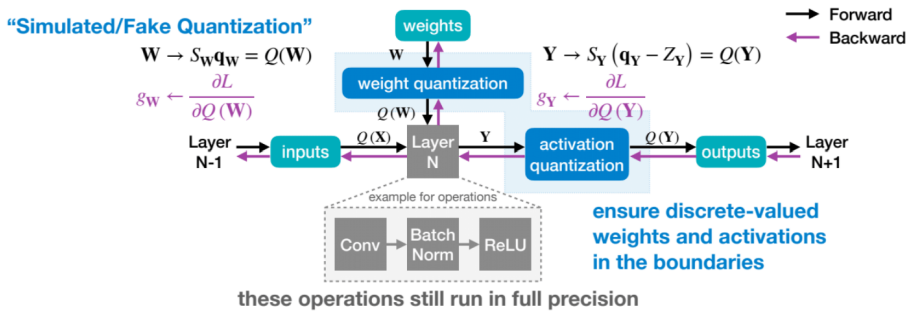
1. Quantization-Aware Training
Quantization-Aware Training(QAT)는 post-training quantization 이후에 수행하는 fine-tuning임.

앞에서 정리한 것과 같이, pruning에서 fine-tuning을 적용하는 것처럼 quantization 이후에도 fine tuning을 수행해 accuracy를 회복시킬 수 있음. 특히 작은 model에 대해서는 quantization만 적용하는 경우 accuracy가 꽤나 떨어지게 되므로 fine tuning의 적용이 중요함.

k-means quantization에 대해서는 앞에서 다룬 것처럼 QAT를 수행할 수 있으므로, 여기에서는 linear quantization에 대한 QAT를 알아보자.

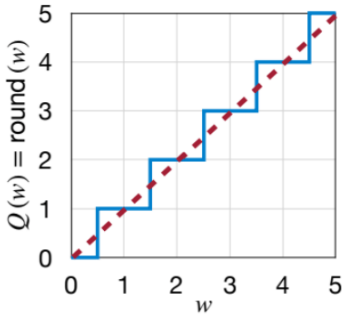
2. Linear Quantization에서의 QAT
linear quantization에 대한 QAT에서는 STE를 활용하여 simulated/fake quantization을 수행함.

QAT에서의 구조는 아래와 같음. forward propagation에서는 weight/activation quantization node를 거쳐 값을 도출함. 이때 Q()는 quantization을 수행하는 function임.



QAT 시에는 full precision(원본) weight를 사용하여 gradient를 적절히 계산하고 최적화를 수행할 수 있음. 이 full precsision weight는 이후 deploy 시에 삭제하여 quantized weight만을 사용함.

forward propagation에서는 단순히 quantize된 weight/activation을 사용하여 출력값을 도출함. 하지만 quantization은 아래의 그래프와 같이 이산적인 출력을 도출하므로 backpropagation에서 quantize된 값으로 gradient를 계산하면 항상 0이 되어 학습이 불가능함.



이에 따라 STE를 사용함. STE(Straight-Through Estimator)는 backpropagation 시에 quantization function을 identitiy function인 것으로 취급하는 기법임. 즉, backpropagation 시에 해당 함수를 없는 것으로 취급하는데, 그 gradient는 원본 W에 대한 gradient와 동일한 것으로 취급함.

$$g_W = \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Q(W)}$$

QAT에서의 이런 forward propagation과 backpropagation 과정을 Simulated/Fake Quantization이라고 함.

3. Neural Architecture Search

3.1. Classic Building Blocks

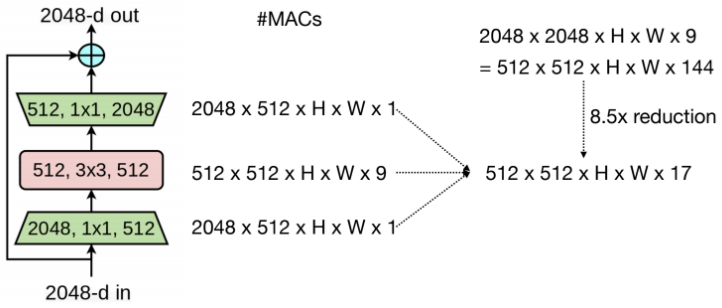
우선 neural architecture에 들어가는 block들을 알아보자. 더 자세한 내용은 '인공지능 기초' 필기를 참고하자.

3.1.1. Bottleneck Block

ResNet50의 Bottleneck block은 1×1 filter를 가지는 2개의 convolution layer와, 그 사이의 3×3 filter를 가지는 하나의 convolution layer로 구성된 residual block임.

bottleneck에서는 1×1 filter를 사용하는 convolution layer에 의해 데이터의 크기를 유지하면서 channel(depth)을 n으로 줄일 수 있음. channel의 개수를 줄인 뒤 다른 3×3 filter convolution을 수행하고, 다시 channels를 늘리면 단순히 하나의 3×3 filter convolution만을 수행할 때보다 computation과 parameter가 대폭 감소함.

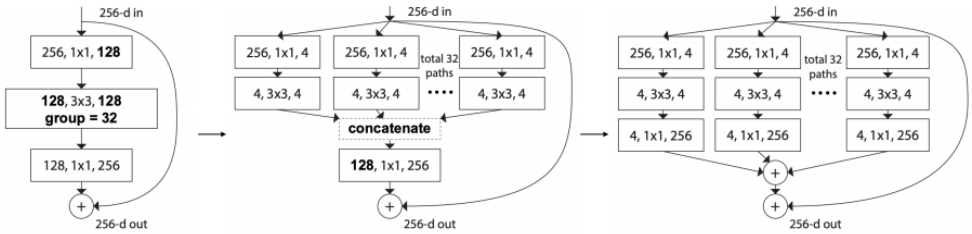
1×1 *filter의 convolution은 padding이 0, stride가 1이고,* 3×3 *filter의 convolution은 padding이 1, stride가 1임. 이에 따라 연산 이후 데이터의 size가 유지되어 residual connection이 가능함.*



3.1.2. Grouped Convolution

ResNeXt(ResNet의 다음 버전)의 Grouped Convolution은 아래 그림과 같이 여러 channel을 가지는 입력 데이터를 channel에 대해 여러 group으로 나누어 각각에 대해 convolution을 수행하는 기법임.

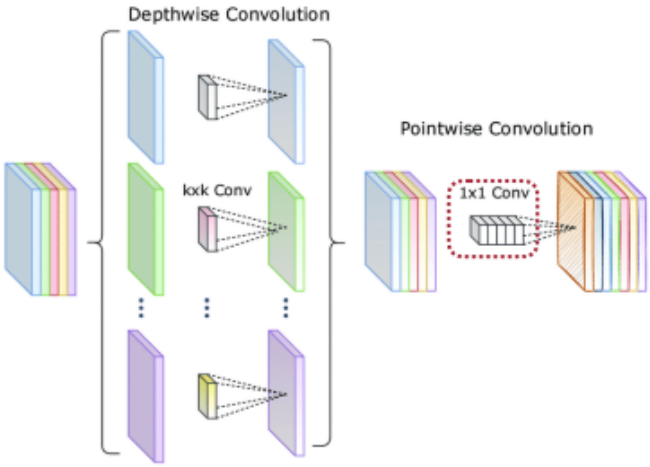
computation 감소를 달성할 수 있음.



3.1.3. Depthwise-separable Block

MobileNet의 Depthwise-separable Block은 depthwise convolution과 pointwise convolution을 함께 사용하는 block임. 이때 depthwise convolution은 spatial(공간적) 관계를 반영하고, pointwise convolution은 channel-wise 관계를 반영함.

computation을 크게 줄이면서 channel 간의 관계를 반영함. 하지만 줄어든 computation에 의해 표현력이 떨어짐.

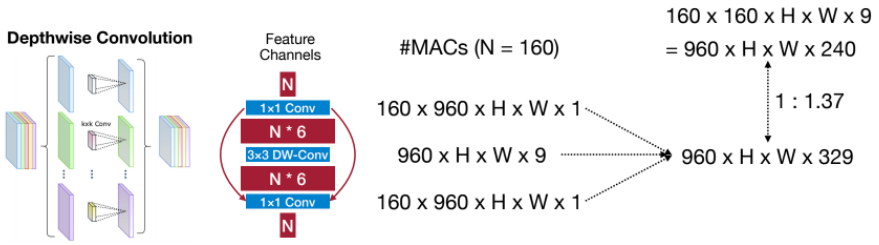


3.1.4. Inverted Bottleneck Block

MobileNetV2의 Inverted Bottleneck Block은 channel의 개수를 줄이는 기존의 bottleneck block과 달리, 반대로(inverted) channel의 개수를 늘리는 동작을 하는 bottleneck block임.

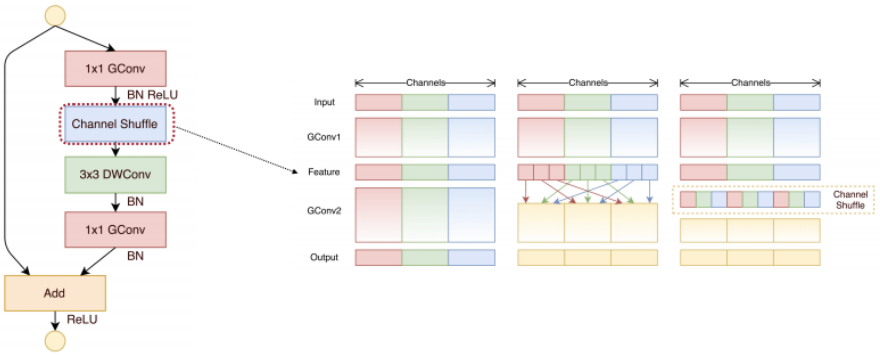
depthwise-separable block 이후 사용하여 떨어진 표현력을 회복시킴. 이를 통해 computation과 paramter 도 적당히 줄이면서 표현력을 확보할 수 있음.

하지만 channel 개수가 늘어나는 만큼 activation memory가 많이 들게 됨. 이는 특히 training 시에 병목이 될 수 있음.



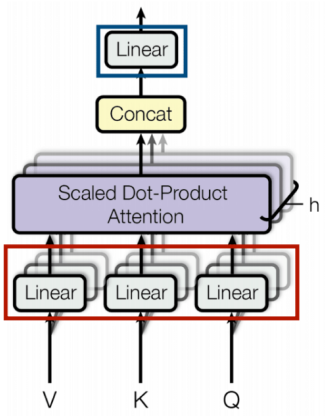
3.1.5. Channel Shuffle

ShuffleNet에서는 group convolution을 수행한 뒤 Channel Shuffle을 사용해 각 group이 가진 channel 이 섞이도록 하고, group convolution을 한번 더 수행해 group 간 상호작용이 가능하도록 함.



3.1.6. Multi-head Self Attention Block

Transformer의 Multi-head Self Attention(MHSA) Block은 아래와 같은 self attention을 각 head에 대해 수행하고, 그 결과를 합치는 block임.



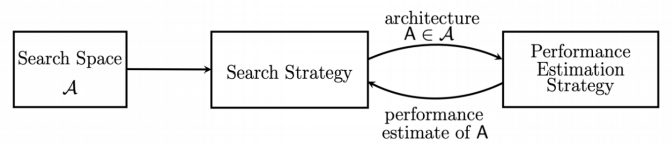
3.2. Neural Architecture Search

3.2.1. Neural Architecture Search

Neural Architecture Search(NAS)는 NN의 아키텍처를 자동으로 찾고 최적화하는 기법임.

제한된 환경에서의 model은 efficiency와 accuracy 사이의 tradeoff 등을 잘 고려하며 설계되어야 함. 이때 고전적인 방식에 따라 manual하게 아키텍처를 설계하는 것은 비용과 시간, 인적 자원이 너무 많이 들고, 실제로는 최적이 아닐 수 있음. 이에 따라 NAS로 automatic하게 최적의 아키텍처를 찾는데, 이렇게 찾은 model은 efficiency, accuracy 모두에 대해 기존의 것들보다 더 성능이 좋다고 함.

NAS는 아래와 같은 동작 과정을 가짐. search space에 존재하는 candidate(후보) 아키텍처들에 대해 search strategy를 적용하여 그 성능을 확인하는 작업을 반복하여 최적의 성능을 보이는 아키텍처를 찾음.



NAS(주로 OFA을 사용하는 것 같음.)를 활용하여 CNN, transformer, 3d vision, GAN(Generative Adversarial Network), pose estimation, quantum ai, LLM 등에서 더 나은 성능의 model을 얻을 수 있 있음.

지금까지는 이미 학습된 model에 대한 compression을 다뤘는데, 여기에서는 제한된 환경에서 동작하는 model에 대한 아키텍처 설계를 다룸.

참고로 Design Space는 DL model 설계 시에 구성할 수 있는 NN 아키텍처 및 hyperparameter들(후보군)의 집합임.

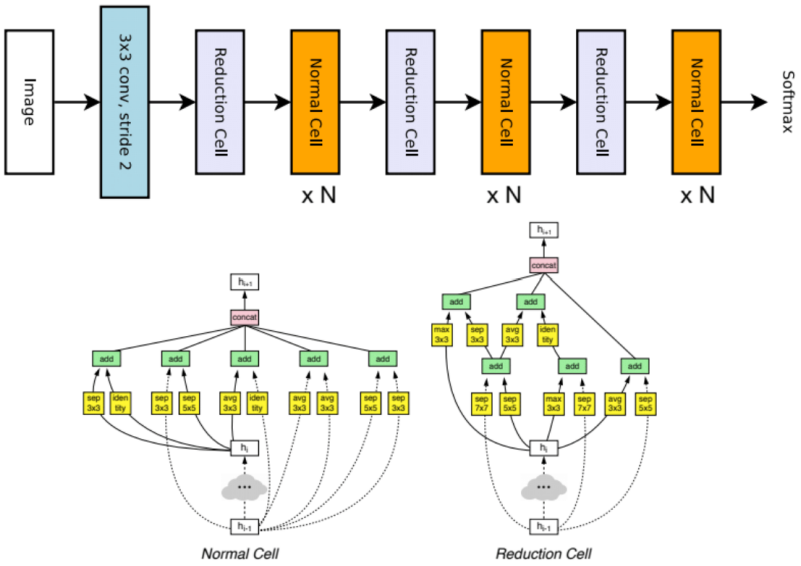
3.2.2. Search Space

Search Space는 NAS에서 선택을 고려하는 candidate NN 아키텍처들의 집합임. search space를 구성하는 방법으로는 아래와 같은 것들이 있음.

1. Cell-level Search Space
Cell-level Search Space는 NN 아키텍처를 cell의 관점으로만 구성하는 방식임. 즉, 사용할 cell들의 구조를 설계하고, 해당 cell들을 반복 사용하여 NN을 구성함. 여기서 Cell은 NN를 구성하는 기본 단위가 되는 block임.

RNN controller를 활용해 cell을 구성할 수 있음. RNN contorller는 아래와 같이 순차적으로 hidden state를 내보내고 5개의 값은 각각 2개의 입력, 2개의 operation, 1개의 combine method를 나타냄. 이 과정을 반복하면 candidate cell들을 생성할 수 있음.

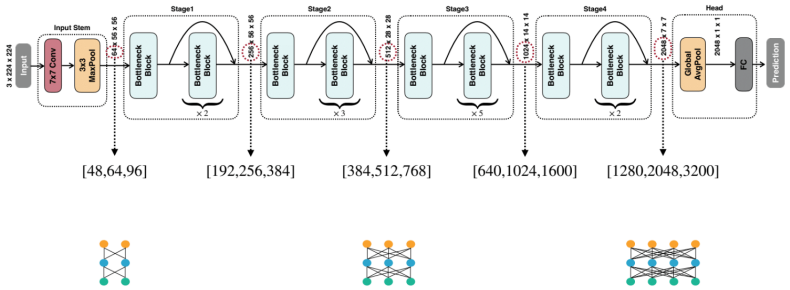
예를 들어, NASnet은 아래와 같이 RNN controller를 사용하여 reduction cell과 normal cell들로 구성됨.



design space가 굉장히 큼.

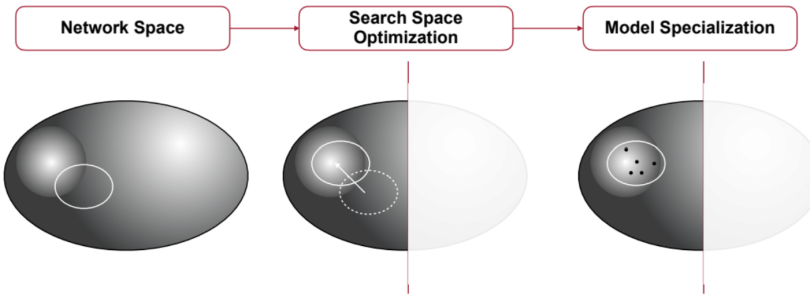
2. Network-level Search Space

Network-level Search Space는 NN 아키텍처를 cell들로 구성된 network 관점에서 구성하는 방식임. 즉, NN의 각 부분에 들어갈 block의 개수(depth), 입력 데이터에 대한 resolution, width(channels)의 dimension, filter의 크기, topology connection(downsample 경로) 등을 조정하여 NN을 구성함.



3.2.3. Search Space Optimization

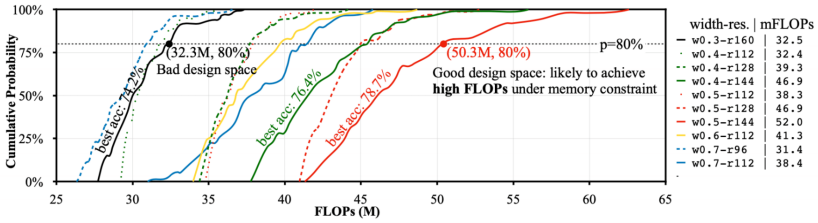
구성된 search space는 그 크기가 굉장히 크기 때문에, 최적화를 통해 높은 성능을 가질 것으로 예측되는 부분으로 search space를 좁혀야(narrow down) 함.



이상적으로는 모든 아키텍처에 대해 학습을 완료하고 accuracy를 측정하는 것이지만 이는 너무 비용이 큼. 즉, model 아키텍처를 학습 없이 평가해야 함.

heuristic하지만 효과적인 방법 중 하나로, 동일한 memory 제약 환경에 대해 내부적으로 높은 flops를

가지는 model이 더 좋은 성능을 가진다고 판단할 수 있음. 이는 대체로 동일한 memory 사용에 대해 더 많은 연산을 수행하는 model의 성능이 더 좋기 때문임. 이 방식으로 search space를 좁히면 단순히 임의의 model을 포함하거나, 큰 search space를 활용하는 것보다 더 높은 accuracy를 도출한다고 함.



3.2.4. Search Strategy

Search Strategy는 NAS에서 search space의 candidiate 아키텍처 중 어떤 것을 선택할지에 대한 전략 또는 최적화 과정임. 아래와 같은 기법들이 있음.

1. Grid Search

Gride Search는 각 변수 값(resolution, width, depth 등)에 변화를 줘서 발생하는 경우들에 대해 성능을 일일이 확인하는 방식임. 아래 그림과 같이 각 grid는 search space에서의 한 아키텍처를 나타냄.

Resolution / Width	1.0x	1.1x	1.2x
1.0x	50.0%	53.0%	**54.9%**
1.1x	51.0%	53.5%	55.4%
1.2x	52.0%	54.1%	56.2%

Satisfies the latency constraint

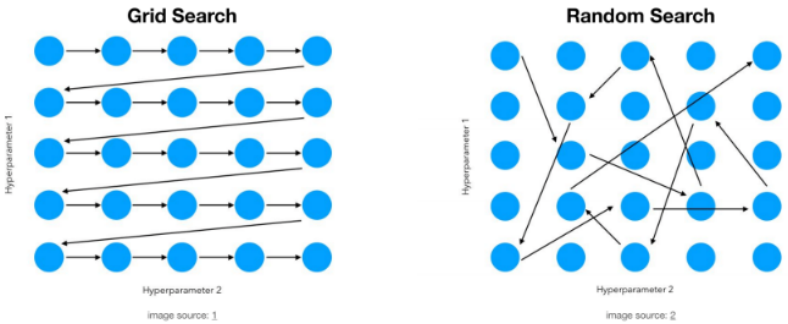
Breaks the latency constraint

이때 각 아키텍처 별로 일부만 학습을 수행한 뒤 성능을 확인하고, 제약을 만족시키는 최선의 아키텍처를 선택할 수 있음.

이는 고전적이고 가장 단순하지만 종종 잘 동작한다고 함.

2. Random Search

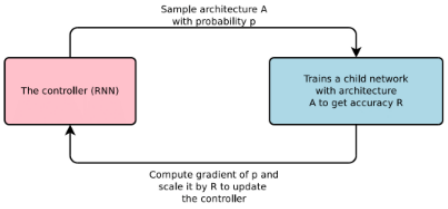
Random Search는 grid search에서 각 변수 값을 순차적으로 변경하며 성능을 확인한 것과 달리, candidate 아키텍처를 무작위로 골라 성능을 확인함.



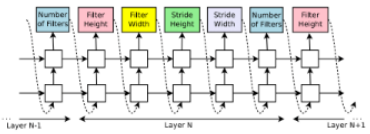
3. 강화학습

아키텍처를 생성하는 RNN controller를 강화학습을 통해 최적화할 수 있음. 즉, 생성된 아키텍처에 대해 성능을 확인하고, 그 결과를 반영함.

실제로 사용하기에 쉽지 않다고는 함.



Overview of RL-based NAS



The RNN controller

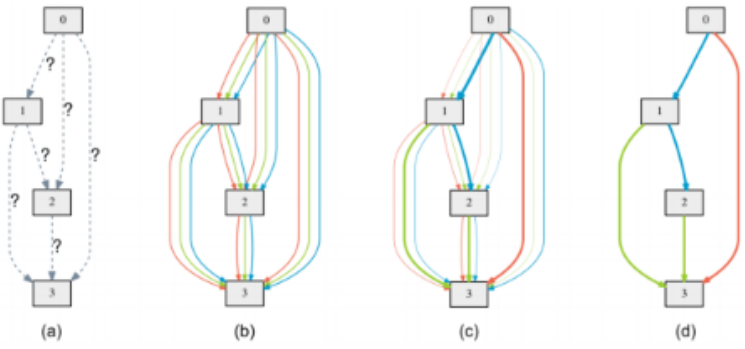
4. Gradient Descent

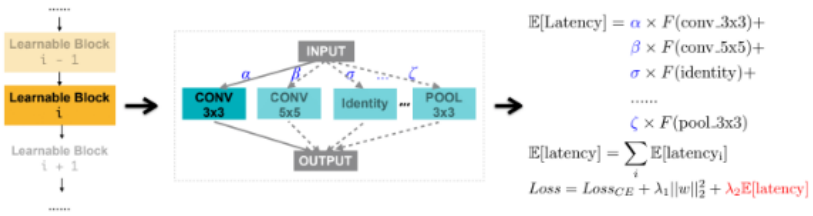
gradient descent를 활용하여 최적의 아키텍처를 찾을 수 있음.

node에서 node로 가는 여러 가지 edge(선택지. operation 등)에 대해 확률 값을 부여하고, 해당 확률에 따른 가중합을 출력으로 함. 이후 gradient descent로 확률 값을 최적화함. 최적화가 끝났으면 가장 높은 확률을 가진 edge를 선택함. 물론 각 경우에 대한 연산을 모두 수행하고 값을 저장해야 하므로 computation과 memory 측면에서는 비용이 많이 듦.

또한 각 방법에 대한 latency를 계산하고 확률 곱한 값을 cost function에 더해 latency를 고려하여 edge를 선택하게 할 수도 있음.

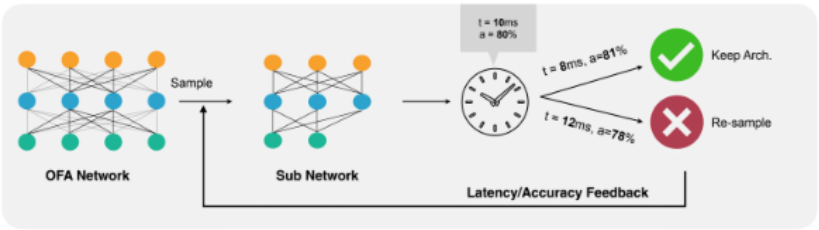
$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})} o(x)$$





5. Evolutionary Search

Evolutionary Search는 생물학에서의 진화 과정을 모방하여, 특정 아키텍처를 생성하고 성능이 적합하면 활용하는 방식임.



3.2.5. Performance Estimation Strategy

Performance Estimation Strategy는 NAS에서 아키텍처 성능을 추정 및 평가하는 전략임. 이때 performance estimation에서는 학습을 최소화하고 accuracy를 정확히 측정할 수 있어야 함.

아래와 같은 방법들이 있음.

1. Train from Scratch
Train from Scratch는 말 그대로 각 아키텍처에 대해 매번 처음부터 학습을 수행하는 방식임.

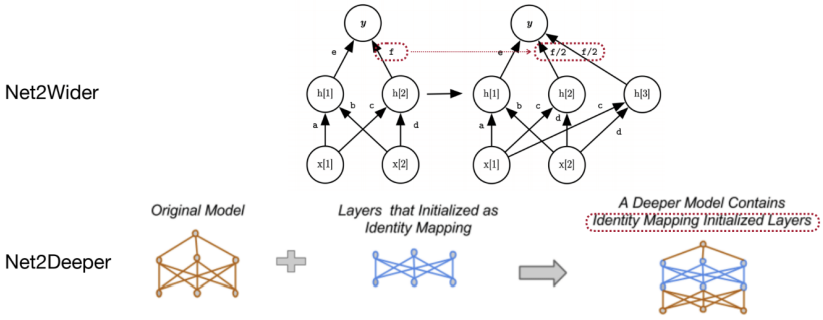
가장 간단하지만 작은 규모에 대해서도 너무 많은 비용이 듦.

2. Inherit Weight
Inherit Weight는 각 아키텍처에 대해 매번 처음부터 학습을 수행하는 대신, 이미 학습된 model의 아키텍처를 수정한 뒤 학습하는 방식임. 즉, 다른 model의 weight를 상속받아 활용함.

inherit weight에서 Net2Wider는 기존의 weight를 반으로 나눠 새로운 neuron을 생성하는 연산(wider 해짐.)이고, Net2Deepr는 기존 network에 identitiy mapping layer를 추가하는 연산(deeper해짐.)임.

이 경우 매번 직접 새로운 아키텍처를 생성하는 대신, 이미 존재하는 아키텍처에 대해 어떤 network transformation action(Net2Wider, Net2Deeper)을 적용할 것인지를 결정함.

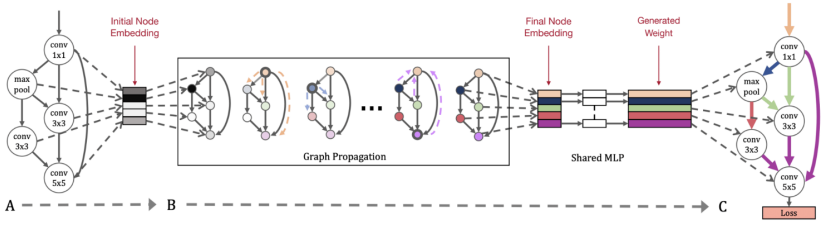
이런 식으로 새로운 아키텍처를 생성하지만 그 수준까지 다시 학습할 필요가 없다. 이 상태에서 시작해 학습시키면 됨.



3. Hypernetwork
Hypernetwork는 weight 예측을 위한 별도의 network임.

search space에서 아키텍처를 가져와 hypernetwork에 넣으면 weight를 생성함. 이후 gradient descent로 hypernetwork를 최적화함.

실제로 잘 사용되지는 않지만 이런 방법도 존재함.



3.3. Efficient/Hardware-aware NAS

앞에서 다룬 것은 단순히 아키텍처를 찾는 방법이었는데, model은 해당되는 환경의 하드웨어에 최적화되어야 하므로 하드웨어를 고려한 NAS 기법에는 어떤 것이 있는지 알아보자. 또한 앞에서 계속 다룬 것처럼 이런 model을 찾는 과정을 자동화하는 것이 중요함.

3.3.1. ProxylessNAS

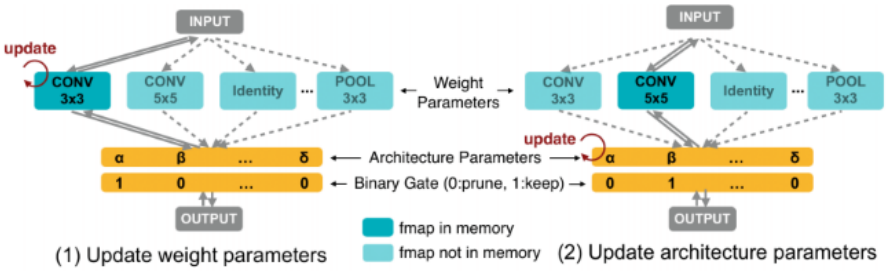
1. ProxylessNAS

ProxylessNAS는 proxy가 아닌 target task/하드웨어에 최적화된 아키텍처를 찾는 방법임.

기존의 NAS는 이론적으로는 획기적이지만 그 과정에 비용이 너무 많이 들었음. 이에 따라 실제 target task/하드웨어 대신 더 쉬운 목표인 proxy task/하드웨어를 목표로 최적의 아키텍처를 찾았음. proxy는 더 작은 데이터셋, 좁은 search space, 적은 epoch, 실제 하드웨어 제약 대신 FLOPS와 parameter 개수 등을 고려하는 식으로 구성되었음. 하지만 당연하게도 proxy에서 잘 동작한다고 target에서도 잘 동작하는 것은 아니기 때문에 최적의 아키텍처를 찾는 데에는 한계가 존재했음. ProxylessNAS는 proxy 대신 target을 사용하도록 하고, 그 비용을 줄인 방식임. 또한 network 수준을 넘어서서 각 block에 대한 설정을 함.

ProxylessNAS는 아래와 같은 구조를 가짐. 모든 candidate path를 사용하여 하나의 network를 구성하고, 각 path에 대한 확률을 나타내는 Architecture Paramter를 사용함. 학습은 path의 weight와 architecture parameter에 대해 각각 수행됨. 이때 모든 path에 대해 한 번에 학습을 수행하는 것은 자원 제약에 따라 불가능하므로, Binary Gate로 0과 1을 지정하여 한 번에 하나의 path(1에 해당하는 path)에 대해서만 학습함.

학습 시에는 architecture parameter를 binarize한 값으로 binary gate를 구성하고, architecture parameter를 freeze한 채로 weight가 학습됨. 이후 weight를 freeze한 채로 architecture parameter가 학습됨. 최종적으로 학습 이후 가장 높은 확률을 가지는 path를 선택함.



간단히 생각해 봐도 모든 아키텍처 각각 고려하는 것보다 더 적은 computation과 memory를 사용함을 알 수 있음.

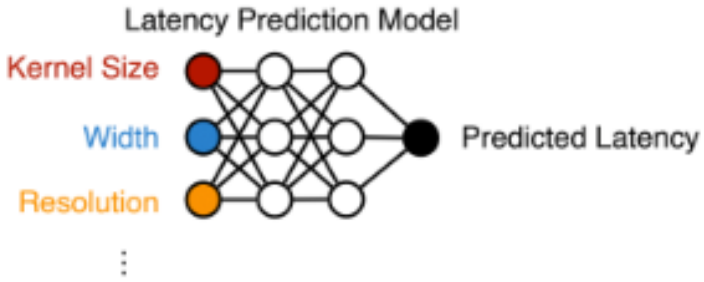
ProxylessNAS를 사용하면 아래와 같이 proxy 대신 target에 대해 최적의 아키텍처를 찾도록 할 수 있음.

• CIFAR-10	• ImageNet
• Small architecture space (e.g. low depth)	• Large architecture space
• Fewer epochs training	• Full training
• Flops and parameter counts	• Profiled Latency

2. Latency 제약 고려

proxy에서는 latency를 FLOPs 등으로 반영하려고 했지만, 앞에서 다룬 것처럼 병렬 처리 등에 의해 MACs와 FLOPs등은 latency를 나타내는 실질적인 지표로 사용되기 어려움. latency를 정확히 반영하기 위해 target 하드웨어에 대해 latency를 직접 측정하는 것이 이상적이기는 하지만 너무 오래 걸리고 비용도 큼.

이에 따라 latency 예측을 위해 latency에 대한 model을 구성하여 사용할 수 있음. 단순하게는 아키텍처의 각 연산에 따른 latency를 반환하는 lookup table을 model로 사용할 수도 있고, 아키텍처와 각 연산의 latency로 구성된 dataset을 활용하여 학습한 linear model을 사용할 수도 있음.



또한 추가로 고려할 문제는, gpu는 layer들끼리의 연산을 합쳐서 병렬 처리를 하기도 하는데 이 경우 layer별 latency 예측값을 전부 더한 것보다 실제 latency가 더 작을 수 있음. 경험적으로는 computation-bound에서는 그런 연산이 잘 안 일어나고, memory-bound에서는 그런 연산이 비교적 자주 일어난다고 함. 필요하면 관련 내용을 더 찾아보자.

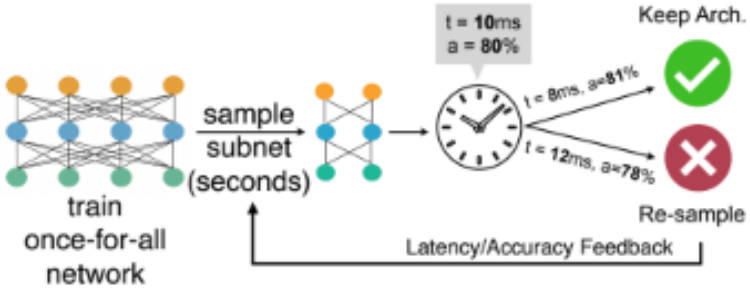
3.3.2. Once-for-All Approach

1. Once-for-All Approach

Once-for-All(OFA) Approach는 하나의 큰 network를 학습시키고 그로부터 각 하드웨어 제약에 최적화된 sub-network들을 추출하는 기법임. 이때 학습을 통해 생성된 큰 network를 OFA Network라고 함.

다양한 platform에 최적화된 model 각각을 설계하고 학습시킨다면 그 비용이 굉장히 커지게 되는데, OFA를 활용하면 학습 한 번(One-shot)으로 여러 환경에 최적화된 network를 얻을 수 있음.

이는 아래의 그림과 같이 OFA network를 학습시키고, 그 sub-network를 추출하여 하드웨어 제약에 부합하는 것을 찾아내는 식으로 동작함. 이때 하드웨어 제약으로는 단순히 device의 차이 뿐만 아니라, 배터리 소모량, memory 사용량 등 여러 기준이 적용될 수 있음.

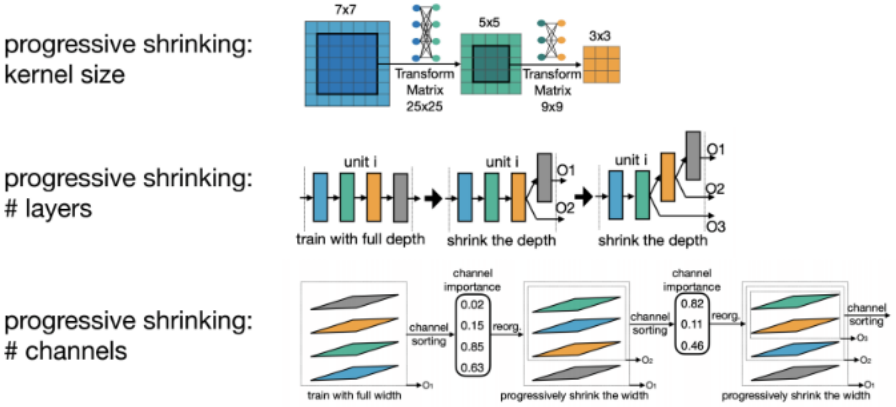


2. Progressive Shrinking

Progressive Shrinking은 OFA network에 대해 가장 큰 network를 점진적으로 축소시키면서 각 sub-network에 대해 학습을 수행하는 기법임. OFA는 progressively shrinking을 통해 OFA network에 대한 한 번의 학습으로 각각의 sub-network들이 잘 동작하도록 구현할 수 있음.

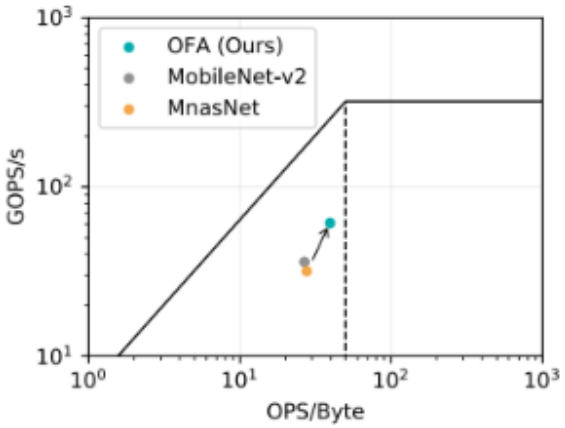
단순히 가장 큰 network에 대해서만 학습을 수행하면 그로부터 추출한 작은 network에서는 성능 저하가 발생할 수 있으므로, OFA approach에서는 점진적으로 학습하는 방식을 사용함.

network의 각 요소(kernel, depth, width)는 학습을 수행하며 점진적으로 수정됨. kernel(filter) size에 대해서는 가장 큰 크기부터 시작해 안쪽 weight로 크기를 줄임. layer(depth)에 대해서는 전체 layer를 사용하다가 끝쪽 layer들부터 줄임. channels(width)에 대해서는 모든 channel을 사용하다가 중요도에 따라 정렬하여 하나씩 제외함.



아래와 같이 각 요소에 대해 순차적으로 progressive shrinking을 적용하며 학습시킴.

참고로 아래와 같이 roofline(이론적 성능 한계) analysis를 수행해보면 OFA designed model들은 기존의 모델들보다 memory에 비해 computation을 많이 활용한다고 함.



3.3.3. Zero-shot NAS

Zero-shot NAS는 학습 없이 성능을 예측하여 최적의 아키텍처를 찾는 기법임. 그 방법으로는 ZenNAS와 GradSign이 있음.

1. ZenNAS

ZenNAS에서는 서로 가까운 두 입력에 대해 출력의 차이가 클수록(입력에 민감함.), 각 layer의 batch normalization에서의 표준편차가 클수록(표현력이 좋음.) model의 성능이 높다고 예측함.

아래와 같이 z_1, z_2를 계산해 더한 것을 score로 함. 이때 z_1을 계산할 때 $x' = x + \epsilon$(ϵ은 매우 작은 수.)이고, f는 해당 아키텍처를 사용하며 가중치를 표준정규분포를 활용해 초기화한 model임.

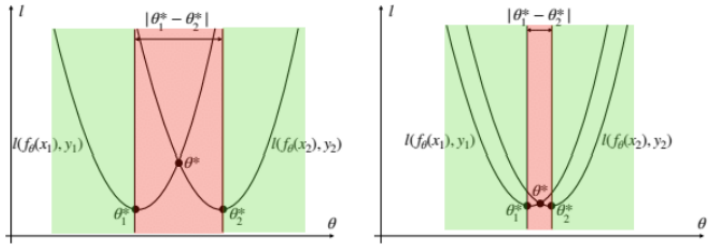
$$z_1 = \log(|f(x') - f(x)|)$$

$$\bar{\sigma}_i = \sqrt{\frac{\sum_j \sigma_{i,j}^2}{c_{out}}}, \quad z_2 = \sum_i \log \bar{\sigma}_i$$

2. GradSign

GradSign은 입력 데이터에 따른 local minima 지점의 차이가 적을수록 좋은 model의 성능이 높다고

예측함. 즉, 비슷한 값의 데이터가 동일한 부호의 gradient를 가지게 되는 경우가 많을수록 좋은 model로 판단하는 것.



(a) Optimization landscape with **sparser** sample-wise local optima corresponding to **worse** $J(\theta^*)$.

(b) Optimization landscape with **denser** sample-wise local optima corresponding to **better** $J(\theta^*)$.

3.3.4. Neural Accelerator Architecture Search

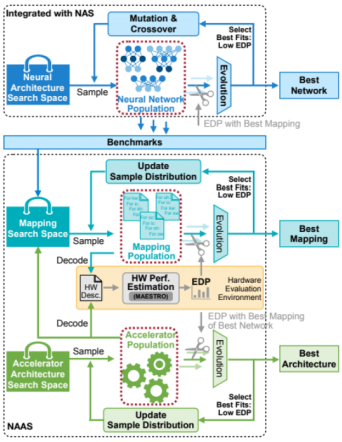
Neural Accelerator Architecture Search(NAAS)는 최적의 accelerator(하드웨어) 아키텍처, NN 아키텍처, 그리고 이 둘 사이의 mapping인 compiler를 찾는 기법임. 지금까지는 제한된 하드웨어 환경에 대한 최적의 NN 아키텍처를 찾으려고 했는데, NAAS에서는 최적의 NN 아키텍처와 하드웨어 아키텍처 모두를 함께 찾음.

아래와 같이 accelerator, NN, compiler 각각에서 고려해야 하는 변수들이 있고, 이를 통해 design space를 생성할 수 있음. 이때 accelerator의 connectivity를 나타내는 부분 등 numeric이 아닌 데이터에 대해서는 나름의 encoding 방식을 적용함.

Key Dimensions		
Accelerator	Local Buffer Size, Global Buffer Size, #PEs	Architectural Sizing
	Compute Array Size, PE Connectivity	Connectivity Parameters
Compiler	Loop Orders, Loop Tiling Size, Dataflow	
Neural Network	#Layers, #Channels, Kernel Size, Bypass	
	(Input / Weight) Quantization Precision	

compiler에서는 어떤 작업을 어떤 순서에 따라 순차적으로 처리할 것인지, 어떤 작업을 병렬 처리할 것인지에 대한 mapping을 정의함.

NAAS는 아래와 같이 evolutionary approach에 따라 accelerator 아키텍처를 생성하고, OFA로 NN 아키텍처를 생성하고, 이 둘 사이의 최적화된 mapping을 생성한 뒤, 그 성능을 측정함.




```
For epoch_naas in range(max_naas_epochs):
  accelerators = NAAS_generate_hardware()
  For hw in accelerators:
    For epoch_ofa in range(max_ofa_epochs):
      networks = OFA_generate_networks(accuracy)
      For nn in networks:
        map = NAAS_optimize_mappings(hw, nn)
        edp = NAAS_get_edp(hw, nn, map)
      OFA_update_optimizer(nn, edp)
      best_nn, best_map, best_edp = OFA_update_best(nn, map, edp)
  NAAS_update_optimizer(hw, best_nn, best_map, best_edp)
```

단순히 NN 아키텍처만을 고려하는 것에 비해 성능을 향상시킬 수 있고, 또한 manual하게 설계했을 때보다 더 좋은 성능의 model을 얻을 수 있다고 함.

4. Knowledge Distillation

4.1. Knowledge Distillation

4.1.1. Knowledge Distillation

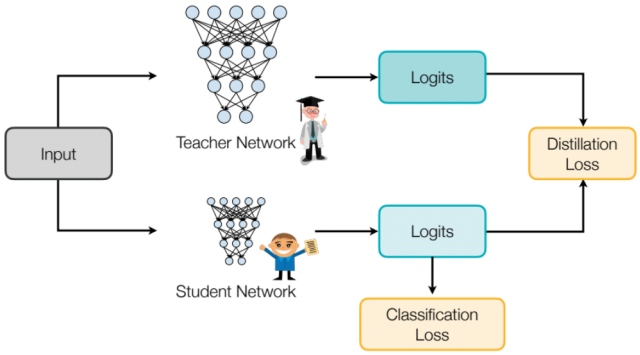
Knowledge Distillation(KD)은 이미 학습된 큰 model(Teacher)을 활용하여 작은 model(Student)의 학습을 돕는 기법임. 즉, teacher와 student의 확률 분포를 맞추는(align, match) 동작을 함.

이는 특히 하드웨어 제약이 엄격한 환경에서 돌아가는 작은 model의 network의 학습 및 성능 개선에 사용될 수 있음.

KD는 teacher와 student에 대해 둘 사이의 loss인 Distillation Loss를 계산하고, gradient descent를 적용해 최적화하는 것으로 구현됨.

기본적으로 아래와 같이 logit(classificaiton에서 softmax를 적용하기 직전의 값.)에 대해 temperature를 적용하는 것으로 구현할 수 있음. Temperature는 아래와 같이 softmax 계산에서 각 값을 나누는 값으로, 출력값의 확률 분포를 조정함. 기본 값은 1이고, 값이 커질수록 확률 분포가 부드러워짐.

$$p(x_i, T) = \frac{\exp(x_i/T)}{\sum_j \exp(z_j/T)}$$



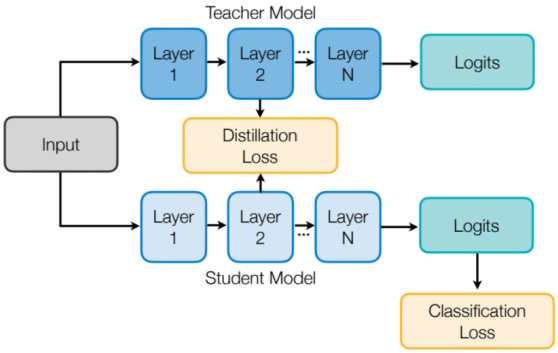
4.1.2. Distillation 대상

KD는 아래와 같은 대상들에 대해 적용될 수 있음.

1. Output Logits
앞에서 다룬 것처럼 KD를 출력값에 대해 적용할 수 있음.

2. Intermediate Tensor
KD를 weight와 activation같은 intermediate tensor에 대해 적용할 수 있음.

아래의 그림과 같이 특정 layer에 대해 weight 값을 비교하여 distillation loss를 계산할 수 있음. 유의할 점은 teacher와 student가 가지는 weight는 그 shape이 다를 수 있다는 것인데, 이 경우 projection matrix(linear transformation)로 shape을 맞춰줘면 됨.



또한 activation(feature)에 대해서도 동일하게 특정 layer에 대해 값을 비교할 수 있음.

3. Gradient
KD를 activation gradient에 대해 적용할 수 있음.

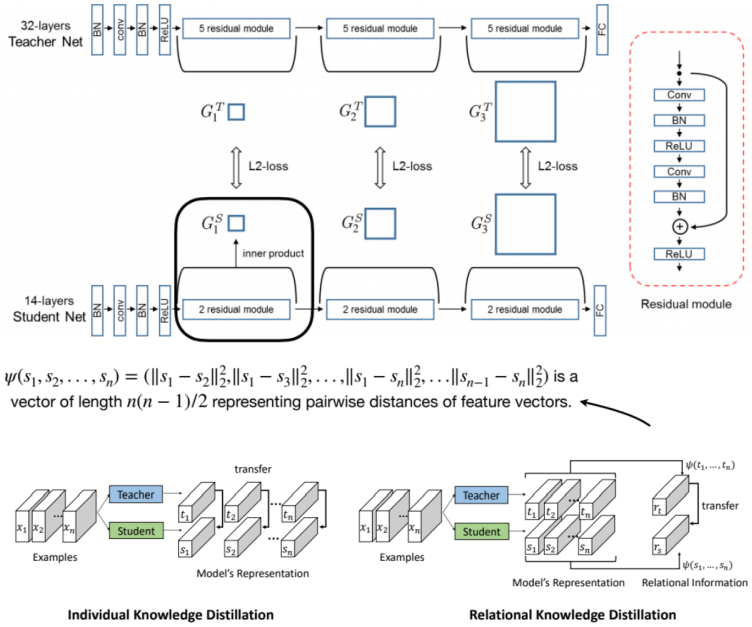
backpropagation 시에는 weight gradient와 activation gradient를 모두 계산할 수 있는데, 이때 teacher와 student는 동일한 입력을 공유하므로 activation gradient를 맞추는 것이 더 쉽다고 함.

4. Sparsity Pattern
KD를 activation에 대한 sparsity pattern에 적용할 수 있음.

직관적으로 teacher와 student가 가지는 각 layer의 activation은 유사한 sparsity pattern을 가짐을 알 수 있음.

5. Relational Information
앞에서 다룬 것과 같이 teacher와 studnet가 각각 가지는 하나의 tensor에 대해서 뿐만 아니라, 여러 tensor의 관계에 대해서 KD를 적용할 수 있음.

특정 부분에 대한 입력 tensor와 출력 tensor의 관계(행렬 곱)를 비교할 수 있음. 또는 서로 다른 입력 데이터끼리의 관계를 볼 수도 있음. 즉, teacher와 student의 값을 일대일로 비교하는 것이 아니라, teacher의 여러 값과 student의 여러 값을 비교함.



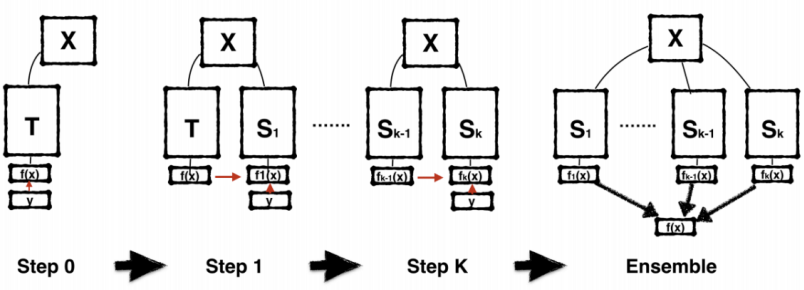
4.1.3. Self/Online Distillation

앞에서 다룬 것과 같이 크고 pre-train(fixed)된 teacher를 학습시키는 방식은 비용이 많이 드는데, self-

/online distillation으로 이를 개선할 수 있음.

1. Self Distillation
Self Distillation은 크고 고정된 model 대신 학습하려는 model을 teacher로 사용하는 기법임.

Born-Again NN은 아래의 그림과 같이 이전 시점에 학습이 완료된 model을 teacher로 활용하는 것을 반복 수행하는 self distillation의 구현임. 즉, 아래와 같이 현재 시점에 대한 classification과, 이전 시점과의 distillation 모두를 수행함. 각 시점에서의 model들은 모두 동일한 아키텍처를 가지고, accuracy는 반복에 따라 계속 계선됨. 또한 각 시점에서의 결과를 assemble하여 최종 결과를 도출할 수 있음.



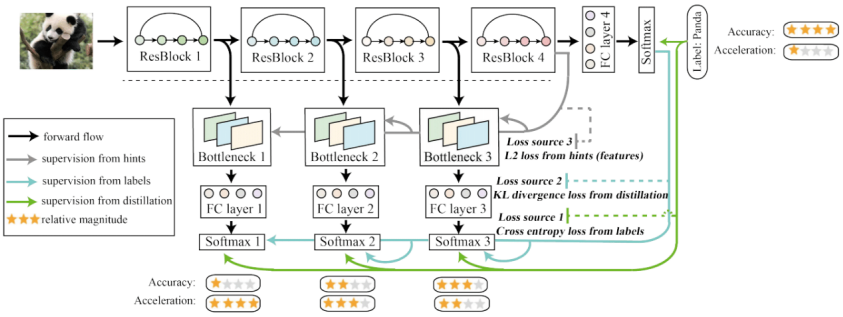
2. Online Distillation
Online Distillation은 미리 학습된 teacher를 활용하여 student를 학습하는 대신, teacher와 student를 동시에 from scratch로 학습시키는 방식임. 이때 두 model은 서로가 teacher이자 student이고, 동일한 아키텍처를 가질 수 있음.

이렇게 여러 model 간의 상호 학습을 Deep Mutual Learning(DML)이라고 함. 두 model에 대해 DML을 적용하면 각각 따로 학습시킬 때보다 성능이 향상된다고 함.

teacher와 student는 아래와 같은 cost function을 가짐. 여기서 KL은 KL Divergence을 말하고, 두 확률분포 사이의 차이를 나타내 distillation을 수행함.

$$\mathscr{L}(S) = \text{CrossEntropy}(S(I), y) + \text{KL}(S(I), T(I));$$
$$\mathscr{L}(T) = \text{CrossEntropy}(T(I), y) + \text{KL}(T(I), S(I)).$$

3. Self + Online Distillation
아래의 그림과 같이 self distillation과 online distillation을 동시에 적용할 수 있음. 즉, 하나의 model에 대해 더 deep한 쪽에 있는 layer의 값을 teacher로 사용하여 shallow한 쪽에 있는 layer에 distillation을 적용함.

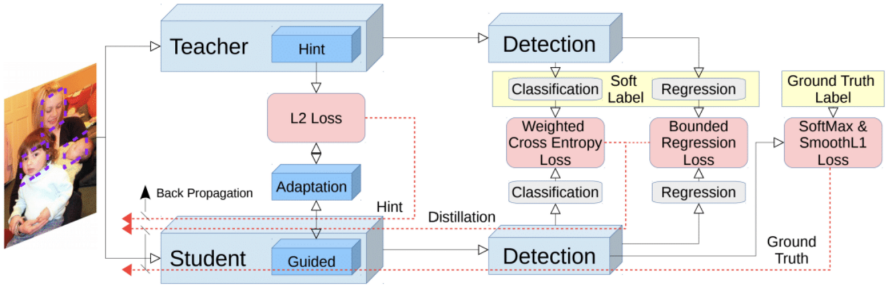


4.2. 분야별 distillation 적용

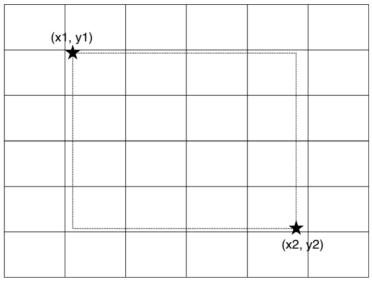
4.2.1. 분야별 distillation 적용

1. Object Detection
Object Detection은 이미지에서 특정 대상의 위치를 찾는 문제임. 아래의 그림과 같이 intermediate 값과 대상에 대한 classification, 위치에 대한 regression에 distillation을 적용할 수 있음.

이때 classification에 대한 distillation으로 계산하는 cross entropy에 weight를 추가한 것은 foreground/background에 따른 불균형을 해소하기 위함임. 이미지 데이터에서는 대체로 foreground(찾으려는 대상)에 비해 background(그 외의 대상, 배경)가 훨씬 많기 때문에, object detection에서 이런 불균형을 고려하지 않으면 model이 background에 편향되어 학습될 수 있음.



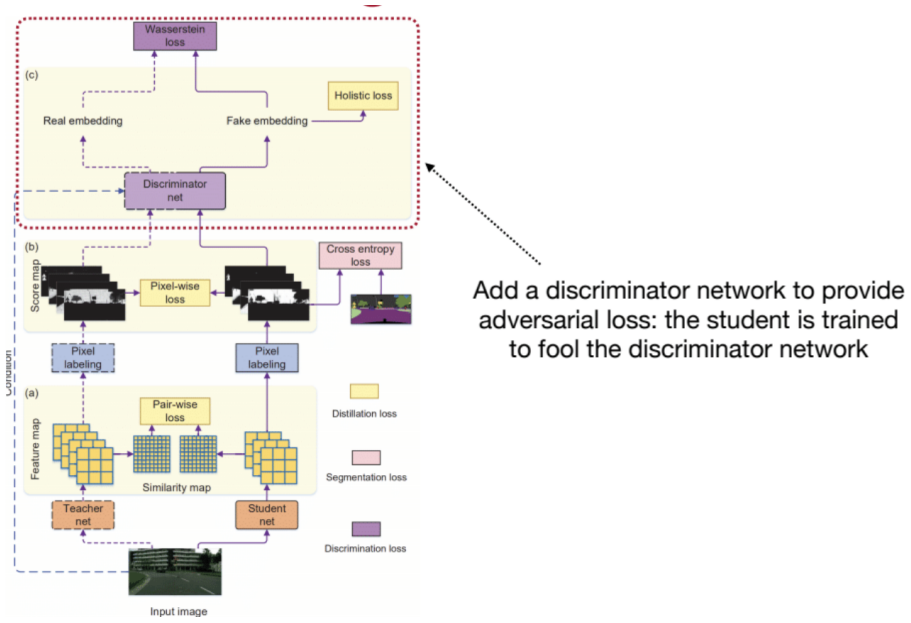
위치 값은 연속형이므로 regression을 적용해야 하는데, 이미지 공간을 격자로 나누어 classification으로도 처리할 수 있음.



2. Semantic Segmentation

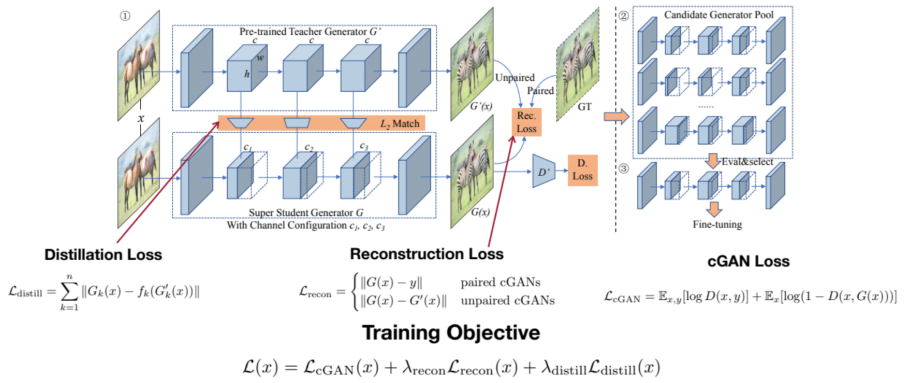
Semantic Segmentation은 pixel 별로 어떤 class에 속하는지를 판단하는 문제임. 즉, pixel-wise prediction임.

아래와 같이 구성할 수 있음. 여기에서 Discriminator Network는 teacher와 student의 출력을 구분하는 network로, student는 discriminator network가 자신의 출력과 teacher의 출력을 구분하지 못하도록 최적화됨.



3. GAN

뒤에서 다루겠지만, GAN에도 아래와 같이 distillation을 적용할 수 있음.

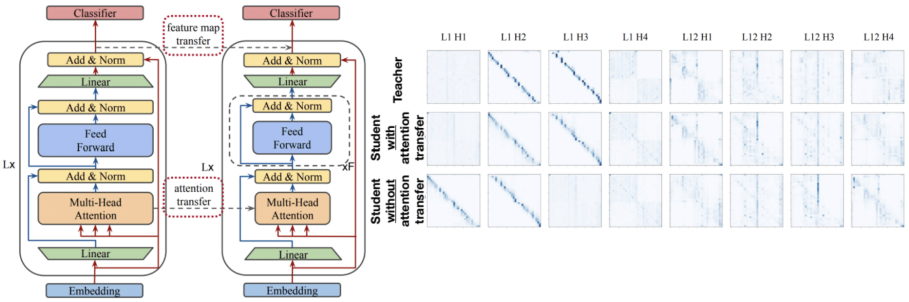


$$\mathcal{L}_{\text{distill}} = \sum_{k=1}^{n} \|G_k(x) - f_k(G'_k(x))\| \qquad \mathcal{L}_{\text{recon}} = \begin{cases} \|G(x) - y\| & \text{paired cGANs} \\ \|G(x) - G'(x)\| & \text{unpaired cGANs} \end{cases} \qquad \mathcal{L}_{\text{cGAN}} = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_x[\log(1 - D(x, G(x)))]$$

$$\mathcal{L}(x) = \mathcal{L}_{\text{cGAN}}(x) + \lambda_{\text{recon}}\mathcal{L}_{\text{recon}}(x) + \lambda_{\text{distill}}\mathcal{L}_{\text{distill}}(x)$$

4. NLP

transformer를 사용하는 경우 feature map뿐만 아니라 attention map도 distill할 수 있음.



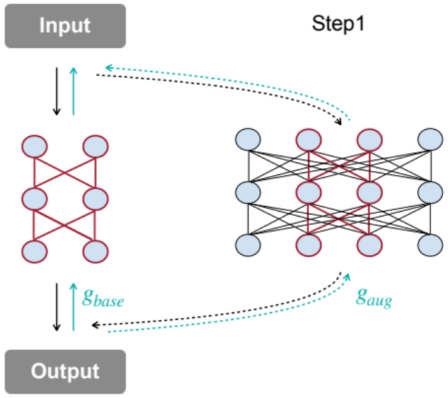
4.3. Network Augmentation

4.3.1. Network Augmentation

Network Augmentation(증강)은 network에 대한 augmentation임.

큰 model에 대해서는 관습적으로 data augmentation과 dropout 등을 적용하여 overfitting이 발생하지 않도록 함. 여기에서 Data Augmentation(증강)은 기존의 데이터로부터 새로운 데이터를 생성하여 overfitting을 방지하는 기법임. 큰 model에 대해서는 이런 기법들이 성능을 높이지만, 작은 model에 대해서는 오히려 성능을 떨어뜨림. 이는 큰 model에서는 overfitting이 주로 발생하지만, 작은 model에서는 underfitting이 주로 발생하기 때문임. data가 아니라 network에 적용하는 augmentation은 underfitting을 해소할 수 있음.

network augmentation은 아래와 같이 기존 network로부터 특정 차원을 확장한 network를 생성하고, model 학습 시에 이 확장된 network를 활용하여 최적화함.



$$\mathcal{L}_{\text{aug}} = \underbrace{\mathcal{L}(W_{\text{base}})}_{\text{base supervision}} + \alpha \underbrace{\mathcal{L}([W_{\text{base}}, W_{\text{aug}}])}_{\text{auxiliary supervision}}$$

작은 model에 network augmentation을 적용하면 성능이 개선되고, 반면 큰 model에 이를 적용하면 overfitting이 발생하여 validation에서 성능이 떨어질 수 있음. 더 큰 model을 생성하여 학습에 활용한다는 점에서 KD와 유사한데, KD보다 더 좋은 성능을 내기도 함.

distillation뿐만 아니라 network augmentation으로도 작은 모델의 accuracy를 높일 수 있음.

5. TinyML

5.1. TinyML

5.1.1. TinyML

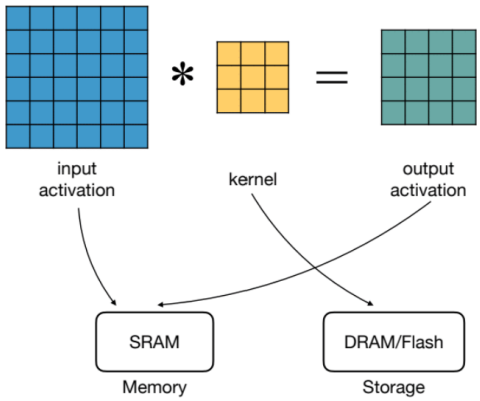
1. TinyML
TinyML은 하드웨어적 제약이 엄격한 edge device(주로 MCU.)에서의 ML임. 특히 MCU(Micro Controller Unit, 마이크로컨트롤러)를 주로 사용하는 Iot 환경에서의 TinyML은 적은 비용, 적은 에너지, 높은 활용성에 따라 그 중요도가 큼.

TinyML은 스마트폰 등 모바일 환경에서 돌아가는 mobile ai에 비해서도 더 엄격한 제약을 만족시켜야 함. 오늘날에 존재하는 기존의 고성능 ai들은 memory와 disk, computation, energy를 너무 많이 사용하므로, 제한된 환경에서는 훨씬 더 많은 efficieny를 가지는 ai를 사용해야 함.

2. Weight/Activation의 저장
weight/activation이 실제로 어디에 저장되는지를 통해 TinyML에서의 하드웨어 제약을 고려할 수 있음.

weight는 그 값을 보존해야 하므로 storage(DRAM/Flash. 주로 flash.)에 저장함. 즉, flash memory는 model size만큼의 공간을 가지고 있어야 함. 이때 추가적인 학습을 하지 않는다면 read only로 저장하기도 함.

activation은 그 값을 매번 계산하므로 memory(SRAM)에 저장함. 당연하게도 input/output activation의 합이 최대인 지점(layer)에서의 peak actvation만큼의 memory 공간을 가지고 있어야 함. 물론 weight도 연산 시에 memory에 올리지만, 부분적인 fetch가 가능하므로 크게 고려하지 않음.



대체로 computation에 비해 memory가 비용이 높은데, 그 중에서도 weight(flash)보다 activation(SRAM) 측면에서의 memory 개선이 어려움.

TinyML은 여러 환경에서 유용하게 활용될 수 있음. 작은 device에서도 유용하지만, 큰 model이 필요한 환경에서도 특정 부분 기능은 efficiency 및 보안 측면에서 TinyML을 활용해 수행하는 것이 좋을 수 있음(visual wake word 등).

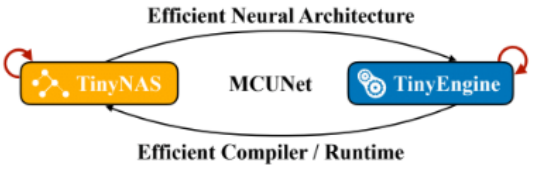
5.2. MCUNet

5.2.1. MCUNet

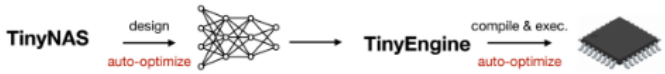
MCUNet은 MCU 등의 환경에서 TinyML을 구현하도록 하는 시스템-알고리즘 공동 설계 프레임워크임. MCUNet은 network 아키텍처 결정을 위한 TinyNAS와, inference scheduling 등을 수행하는 추론 엔진인 TinyEngine으로 구성되고, 이를 통해 자동으로 최적화를 수행함.

추론 엔진(Inference Engine, Library)은 학습된 model을 활용해 실제로 inference를 수행하는 부분으로, compiler/runtime 등을 포함함. compiler는 언어에 대한 compiler와 유사하게 model을 해당 환경의 하드웨어에 실행 가능한 형태로 변환하는 부분이고, runtime은 컴파일된 코드를 실제로 실행하는 부분임.

TinyEngine은 뒤에서 설명할 parallel computing 기법들과 inference optimization을 활용해 효율적으로 inference를 수행함.



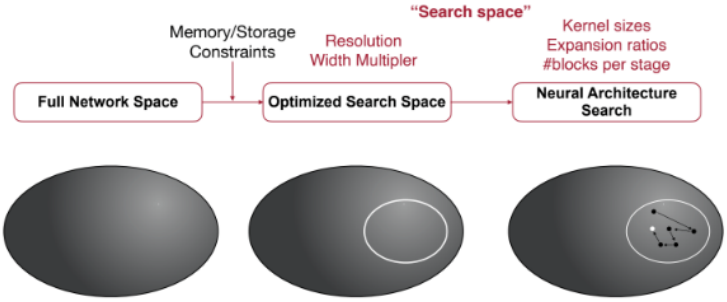
(c) *MCUNet*: system-algorithm co-design



5.2.2. TinyNAS

1. TinyNAS
TinyNAS는 MCU 등의 환경에서 돌아가는 model에 대한 NAS임.

TinyNAS는 아래와 같이 2개의 stage를 가짐.
1) search space 최적화
2) Neural architecture search



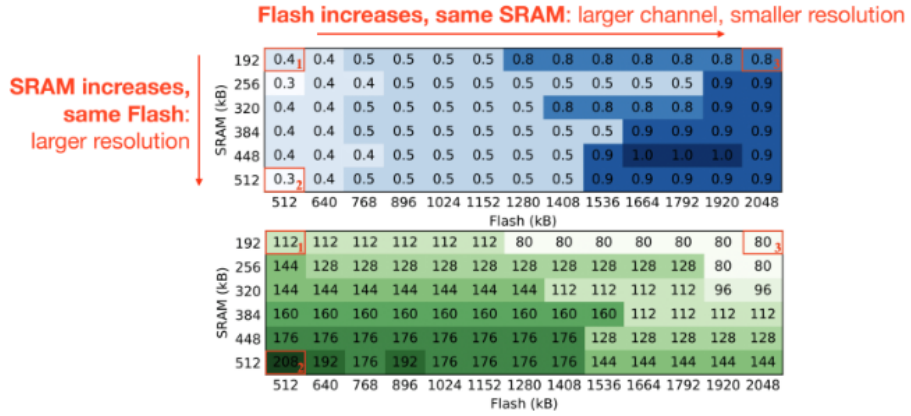
2. Search Space 최적화

TinyNAS는 search space를 자동으로 최적화함. search space가 잘 최적화해야 효율적으로 성능 좋은 아키텍처를 얻을 수 있음.

이미 존재해는 다른 search space를 그대로 가져와 사용할 수도 있지만, MCU와 같이 극도로 제한된 환경에 대해서는 가장 작은 버전도 제약을 만족시키지 못할 수 있다는 문제가 있음. 이에 따라 가져온 search space를 scaling하여 최적화한 뒤 활용함. 물론 gpu 등에 대해서는 scale-up을 적용해 성능을 개선하지만, MCU에 대해서는 scale-down을 적용해 성능을 개선할 수 있음.

resolution과 width(channels. width multiplier를 곱하는 식으로 구현됨.)에 대해 scaling을 적용하면 SRAM, flash 자원에 부합하는 search space를 얻을 수 있음. 이때 resolution은 activation에 대한 것, width는 weight에 대한 것임.

아래와 같이 flash, SRAM의 크기 변화와 resolution, width는 연관되어 있음. SRAM이 커지면 resolution이 증가하고, flash가 커지면 width가 증가하고 resolution이 감소함. 여기에서 flash가 커지면 더 많은 weight를 저장할 수 있어 width가 증가하지만, activation 크기는 유지해야 하므로 resolution이 감소하는 것임.



flash, SRAM 자원에 부합하는 search space들 중, 앞에서 다룬 것과 같이 해당되는 model이 높은 FLOPs를 가지는 design space를 선택함.

3. Neural Architecture Search

search space를 찾은 뒤에는 OFA를 사용해 one-shot으로 아키텍처를 찾음. 즉, supernetwork를 학습한 뒤 그 subnetwork를 추출하고 fine tuning하여 충분한 성능의 아키텍처를 얻음.

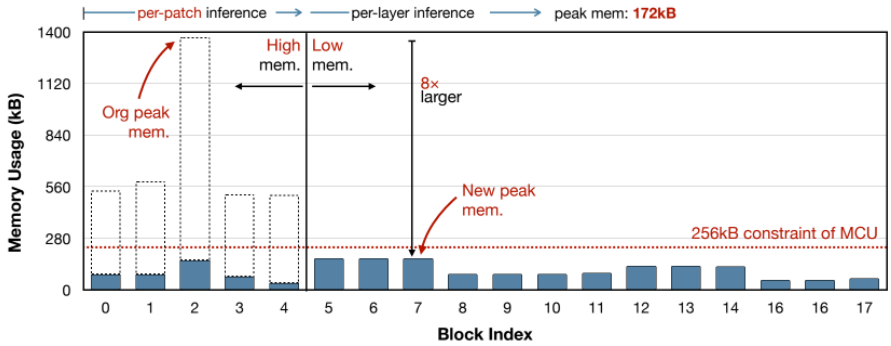
TinyNAS는 수작업으로 설계한 아키텍처에 비해 memory 사용량을 줄이고, block별 memory 사용량을 균일하게 좁혀서 낭비를 최소화한다고 함.

5.2.3. Patch-based Inference

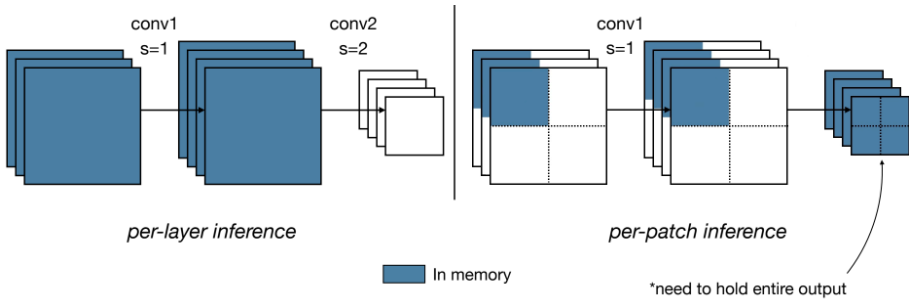
1. Patch-based Inference

Patch-based Inference는 activation을 여러 개의 patch로 나누어 처리하는 기법으로, MCUNetV2의 핵심 아이디어 중 하나임. TinyEngine에서는 이 기법을 사용하여 inference scheduling을 수행함.

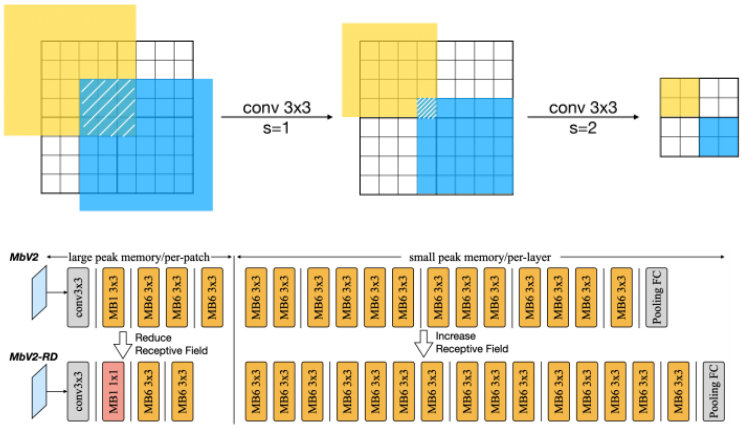
flash에 대해서는 앞에서 다룬 것처럼 model size를 줄여서 최적화할 수 있고, SRAM에 대해서는 peak activation(이게 bottleneck임.)을 줄여 최적화 할 수 있음. 그런데 아래 그림과 같이 각 block에서의 (CNN 기준) input/output activation 합을 확인해 보면 앞쪽 layer에서의 memory 사용량이 훨씬 많음. 이에 따라 patch-based inference에서는 앞쪽 layer에서의 activation을 줄이는 동작을 함.



아래와 같이 각 layer에서 모든 부분에 대한 계산을 한 번에 수행하는 대신, 연산을 여러 부분(patch)으로 나눠서 수행함. 이에 따라 patch의 개수에 비례하는 메모리를 줄일 수 있음. 물론 마지막 결과를 저장하고 있어야 하기 때문에 무작정 줄일 수 있는 것은 아님. 이런 patch 별 연산을 앞쪽 layer에 적용하여 activation을 줄임.



하지만 이와 같은 작업은 conv 연산에 따라 겹치는 부분이 생겨 latency overhead가 존재함. 즉, 동일한 연산을 재수행하게 될 수 있음. 특히 recepive field가 커질수록 겹치는 부분이 많으므로 latency가 커짐. 이에 따라 MCUNetV2에서는 앞쪽 layer의 convolution layer를 없애거나 수정해 receptive field를 줄이고, 이에 대한 보상으로 뒤쪽 layer에 conv layer를 추가하는 것으로 구현되었음.

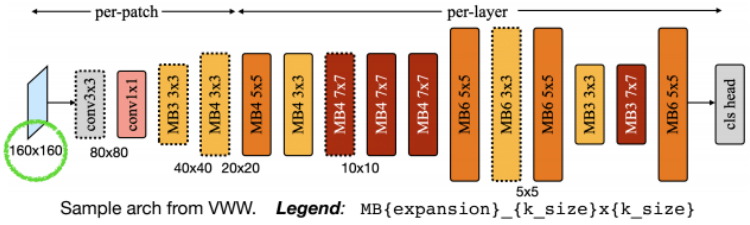


이런 기법을 활용하여 MCUNetV2는 MCUNet에 비해 훨씬 적은 SRAM으로 더 뛰어난 성능을 보인다고 함.

2. MCUNetV2 아키텍처

patch-based inference에 따라 MCUNetV2는 아래와 같은 아키텍처를 가짐. 앞쪽 layer들에서는 작은

kernel을 사용하여 latency overhead를 줄였고, 중간 부분에서는 bottleneck block에서 적은 expansion ratio을 사용하여 peak memory를 줄임. 이후 표현력을 다시 확보하기 위해 뒤쪽 layer에서는 큰 kernel과 expansion ratio를 사용함.



Sample arch from VWW. ***Legend:*** `MB{expansion}_{k_size}x{k_size}`

object detection에서 accuracy를 유지하려면 높은 resolution을 처리할 수 있어야 하고, 이에 따라 peak activation이 커져 큰 SRAM이 필요하게 됨. 이런 경우에 patch-based inference가 더욱 유용함.

참고로 audio 데이터는 2차원 데이터(time, frequency)로 변환하여 visual 데이터처럼 처리할 수 있음. 특히 audio 데이터에는 locality가 존재하므로 CNN을 사용하면 좋은 성능을 보일 수 있음.

anormaly detection 등에도 MCUNet을 활용할 수 있음. Autoencoder는 입력과 동일한 출력을 도출하는 mode인데, 이를 활용해 정상적인 입력과 비정상적인 입력을 분간할 수 있음. 정상적인 입력에 대해 입력과 동일한 출력을 도출하도록 학습시키면, 이후 특정 데이터가 정상적이지 않을 때 입력과 동일하지 않은 출력이 도출됨.

5.3. Parallel Computing 기법들

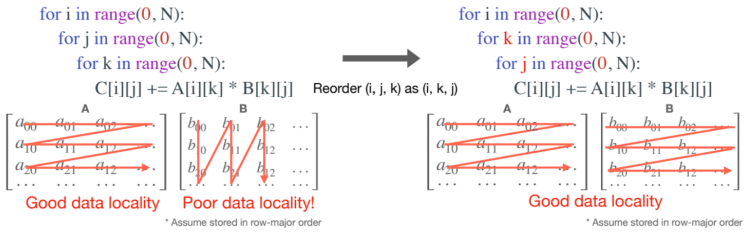
5.3.1. Loop Optimization

Loop Optimization은 반복문에 적용하는 optimizaiton으로, 아래와 같은 기법들이 존재함.

1. Loop Reordering

Loop Reordering은 다중 loop에 대해 loop 순서를 적절히 재배치하여 data locality를 최대한 활용하도록 하는 기법임.

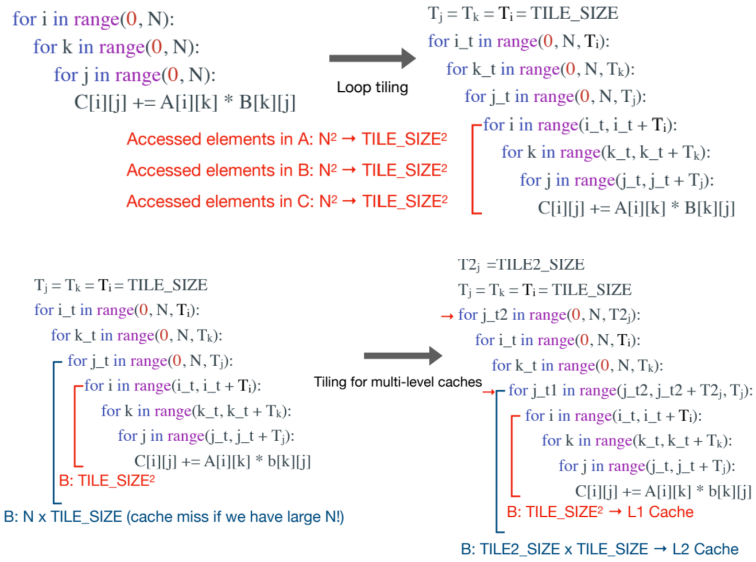
특히 아래와 같이 행렬 곱을 할 때 locality를 고려하지 않으면 cache miss가 자주 발생함. 이 경우 loop 순서만 바꿔도 처리 속도를 크게 개선할 수 있음. 물론 이때 순서를 재배치하면 행렬 C에 대해서는 locality를 활용하지 못하는데, 실제로는 행렬 B에서의 개선이 c에서의 overhead를 훨씬 상회한다고 함.



2. Loop Tiling

Loop Tiling은 처리하는 행렬(데이터)을 더 작은 단위인 tile로 나누어 cache에 올리는 기법임. cache보다 작은 크기의 tile을 사용하여 cache miss를 줄일 수 있음.

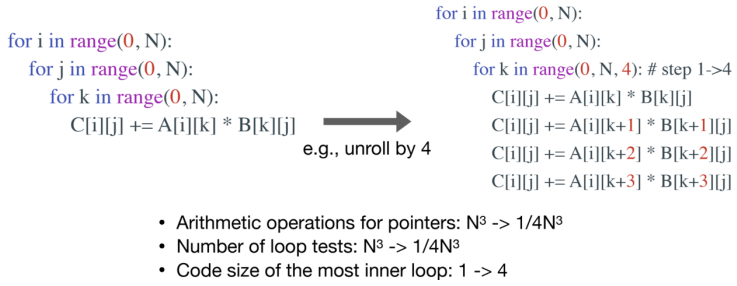
아래와 같이 single level cache의 크기에 맞춰 tiling을 적용할 수 있고, multi-level cache에 대해서도 적용할 수 있음. 이에 따라 cache miss가 대폭 제거되어 훨씬 빨라짐.



3. Loop Unrolling

Loop Unrolling은 loop 내부의 instuction을 unroll하여 branch predicetion, loop 변수 연산, 조건문 처리 등의 overhead를 줄이는 기법임. 물론 이에 따라 코드 자체의 길이가 길어진다는 tradeoff가 존재함.

아래와 같이 연산량을 줄일 수 있음.



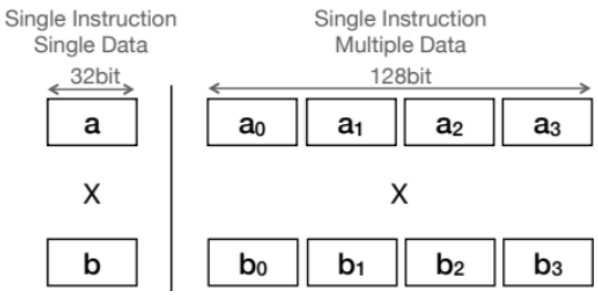
- Arithmetic operations for pointers: N^3 -> $1/4N^3$
- Number of loop tests: N^3 -> $1/4N^3$
- Code size of the most inner loop: 1 -> 4

5.3.2. SIMD programming

SIMD(Single Instruction Multiple Data)는 하드웨어적인 지원을 통해 하나의 instuction으로 여러 개의 데이터를 병렬 처리하는 기법임. 주로 사용되는 SIMD instuction set으로는 SSE와 NEON이 있음.

SIMD는 vector register와 vector operation을 활용함. Vector Register는 여러 개의 데이터를 담는 register이고, Vector Operation은 vector register에 대해 수행하는 연산임.

SIMD를 사용하면 여러 데이터에 대한 병렬 연산(data-level parallelism) 및 instruction decoding 시간 단축으로 벡터 연산에 대한 성능이 향상됨.



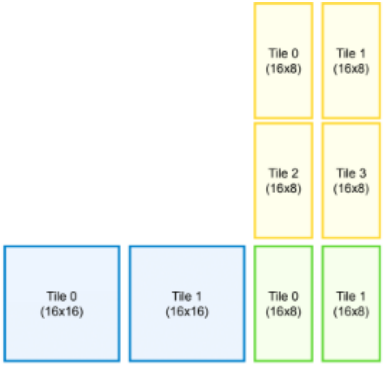
SSE와 NEON에서는 SIMD programming을 위해 intrinsic function을 제공함. Intrinsic Function은 c/c++등 고수준 언어에서 하드웨어의 SIMD instruction을 직접 호출할 수 있도록 컴파일러가 제공

하는 함수임. 하드웨어 조작을 위해 어셈블리어를 사용하는 대신 아래와 같이 이를 활용하여 SIMD programming을 할 수 있음.


```
preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
  for (j = 0; j < C->column; j++) {
    float accumulators[4] = {0, 0, 0, 0};
    __m128 *acc = (__m128*)accumulators; // initialize four 32-bit accumulators
    for (k = 0; k < A->column; k += 4) {
      // val[0:4] = A[i][k:k+4] * A[j][k:k+4]
      __m128 val = _mm_mul_ps(_mm_load_ps(&A[i][k]), _mm_load_ps(&transpose_tmp[j][k]));
      // accumulators[0:4] = accumulators[0:4] + val[0:4];
      *acc = _mm_add_ps(*acc, val);
    }
    c[i][j] = accumulators[0] + accumulators[1] + accumulators[2] + accumulators[3];
  }
```

```
preprocessing(); // Initialize A, B, C and transpose B as transpose_tmp

for (i = 0; i < C->row; i++)
  for (j = 0; j < C->column; j++) {
    float accumulators[4] = {0, 0, 0, 0};
    float32x4_t *acc = (float32x4_t*)accumulators; // initialize four 32-bit accumulators
    for (k = 0; k < A->column; k += 4) {
      // val[0:4] = A[i][k:k+4] * A[j][k:k+4]
      float32x4_t val = vmulq_f32(vld1q_f32(&A[i][k]), vld1q_f32(&transpose_tmp[j][k]));
      // accumulators[0:4] = accumulators[0:4] + val[0:4];
      *acc = vaddq_f32(*acc, val);
    }
    c[i][j] = accumulators[0] + accumulators[1] + accumulators[2] + accumulators[3];
  }
```

또한 tensor core에서는 행렬 곱 등에서 데이터의 크기가 하나의 intrinsic이 처리할 수 있는 것보다 큰 경우, 데이터를 여러 tile로 나눈 뒤 각 tile에 대한 연산을 순차적으로 수행함.

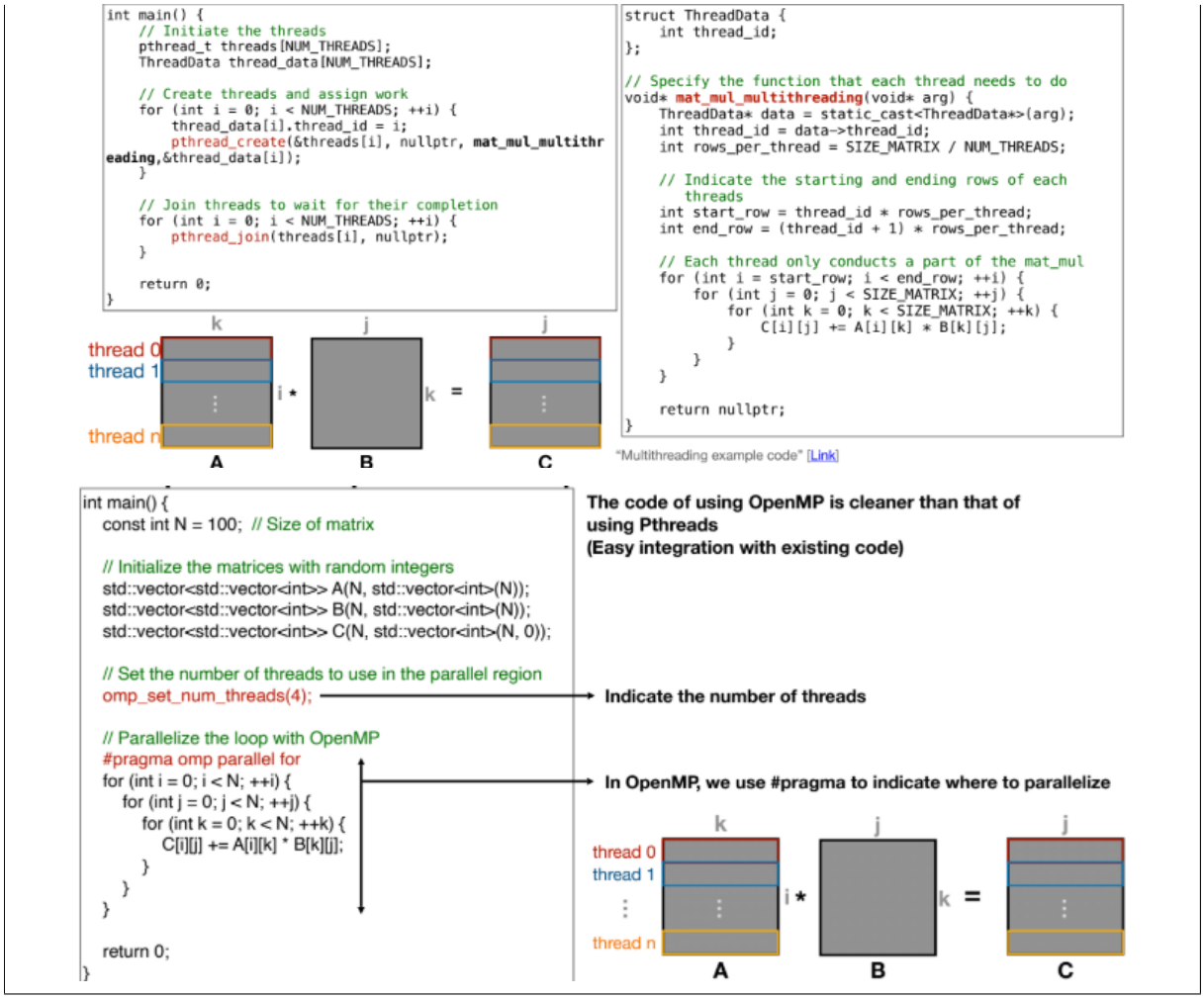


Note: The order of applying these four MMA intrinsics does not matter.

5.3.3. Multithreading

multithreading으로 병렬 처리를 구현할 수 있음. c/c++에서는 Pthreads와 OpenMP를 활용할 수 있음.

아래와 같이 Pthreads와 OpenMP를 사용할 수 있음. Pthreads는 비교적 복잡하지만 더 세밀한 제어가 가능하고, OpenMP는 간단하게 사용할 수 있지만 컴파일러에 의해 자동으로 제어됨. OpenMP에서는 #을 활용한 구문을 작성하여 컴파일러에게 multithreading 처리를 하도록 지시할 수 있음.



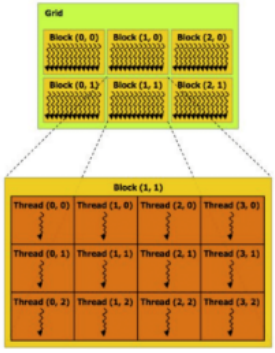
5.3.4. CUDA programming

1. CUDA

CUDA(Compute Unified Device Architecture)는 c/c++ 등 고수준 언어에서 gpu를 활용할 수 있도록 Nvidia에서 개발한 소프트웨어 스택임. CUDA는 gpu를 활용하는 데에 필수적인 툴임.

CUDA에서 host는 cpu를, device는 gpu를 말하고, device에서 돌아가는 코드를 포함하는 함수를 Kernel이라고 함. 이때 kernel을 gpu에서 돌리기 위해 kernel call을 실행하는데, 이는 kernel lanunch overhead 등 overhead가 존재해서 가능하면 한 번의 kernel call에서 여러 작업을 처리하도록 구현한다고 함.

CUDA는 thread에 대한 계층 구조를 사용하여 효과적인 병렬 처리를 하도록 도움. 이 계층 구조는 아래와 같이 여러 thread를 포함하는 block과, 여러 block들을 포함하는 grid로 구성됨.



Regular application thread running on CPU (the "host")

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

CUDA kernel definition

```
// kernel definition (runs on GPU)
__global__ void matrixAdd(float A[Ny][Nx], float B[Ny][Nx],
float C[Ny][Nx])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  C[j][i] = A[j][i] + B[j][i];
}
```

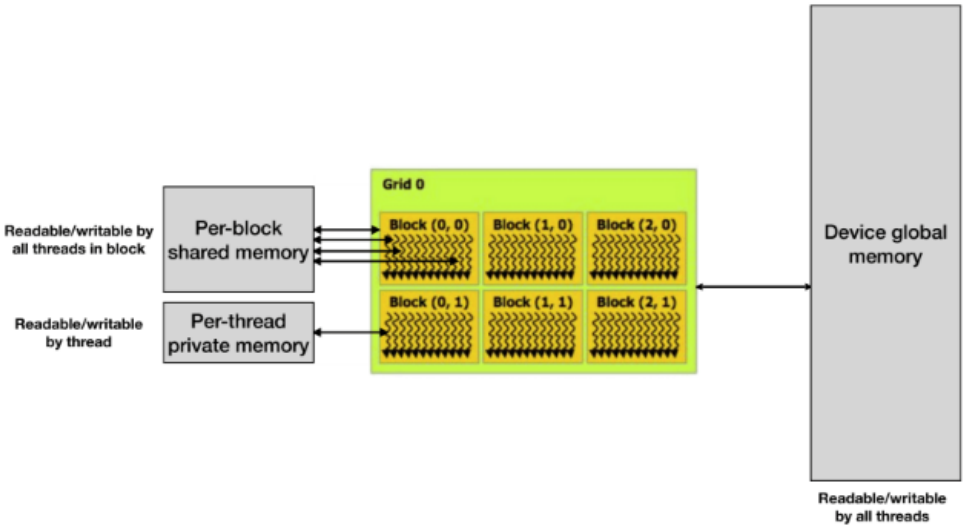
→ Kernel code runs on CUDA device: Parallel execution

→ Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

2. Memory Model

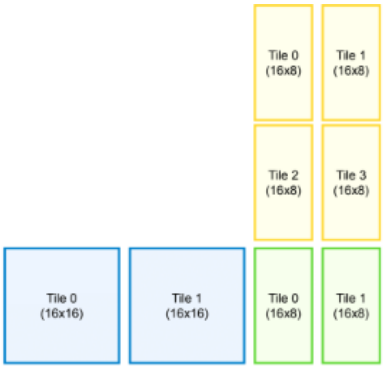
device는 host와 분리된 별도의 memory를 가짐. 아래와 같이 device에 memory를 할당하고, host의 데이터를 옮길 수 있음. 또한 kernel에서 접근할 수 있는 memory는 Device Global Memory, Per-block Shared Memory, Per-thread Private Memory로 총 3가지임.

```
float* A = new float[N]; // allocate buffer in host memory

// populate host address space pointer A
for (int i=0;i<N;i++)
  A[i] = (float)I;

int bytes = sizeof(float) * N;

float* deviceA;
cudaMalloc(&deviceA, bytes)  // Allocate buffer in device address space

// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);
```



또한 Nvidia gpu의 tensor core는 tensor 연산에 대한 더 높은 throughput과 다양한 data type을 지원하고, 처리하는 데이터의 크기가 커질수록 기존 core에 대한 성능 향상이 커짐.

또한 tensor core에서는 행렬 곱 등에서 데이터의 크기가 하나의 intrinsic이 처리할 수 있는 것보다 큰 경우, 데이터를 여러 tile로 나눈 뒤 각 tile에 대한 연산을 순차적으로 수행함.



Note: The order of applying these four MMA intrinsics does not matter.

5.4. Inference Optimization

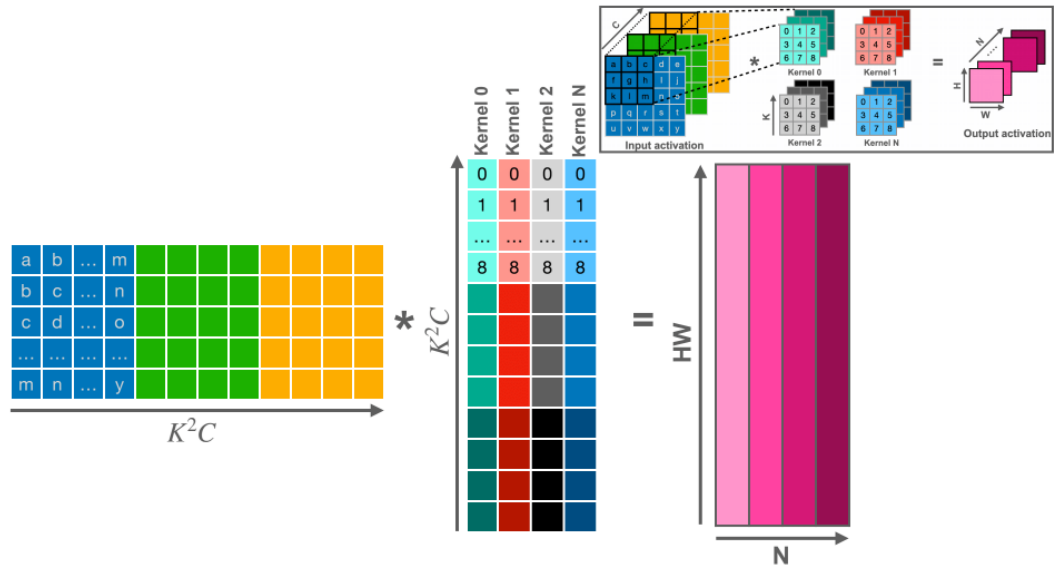
앞에서 pruning, quantization, NAS, distillation 등의 기법들은 NN에서 발생하는 수학적인 연산 자체를 수정하여 efficiency를 개선함. 반면 여기서의 최적화는 연산을 수정하지 않고 실제 inference 시에 성능 개선에 대한 것임. 즉, inference 과정이 구현 방식을 개선해 efficiency를 높임.

5.4.1. Im2col Convolution

Im2col Convolution은 convolution 연산을 행렬 곱으로 변환하여 계산하는 기법임.

행렬 곱은 intrinsic 관점에서도, building block 관점에서도 가장 기본적인 연산인데에 반해, covolution 연산은 행렬 곱과는 다른 형태의 연산임. 이에 따라 Im2col convolution에서는 입력 데이터를 재정렬하여 convolution 연산을 행렬 곱으로 계산함.

아래와 같이 input activation을 펼친 뒤 재구성한 행렬끼리의 곱셈으로 convolution 연산을 수행할 수 있음.

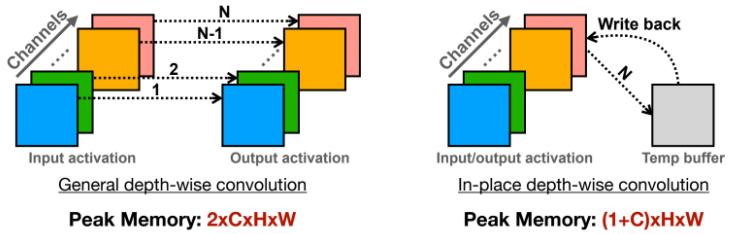


물론 겹치는 부분에 따른 추가적인 memory 사용이 있을 수 있음.

5.4.2. In-place Depthwise Convolution

In-place Depthwise Convolution은 buffer를 추가로 하나 사용하여 depthwise convolution을 수행하고, 그 결과를 입력 데이터 memory 중 해당 channel에 해당하는 부분에 저장하는 기법임.

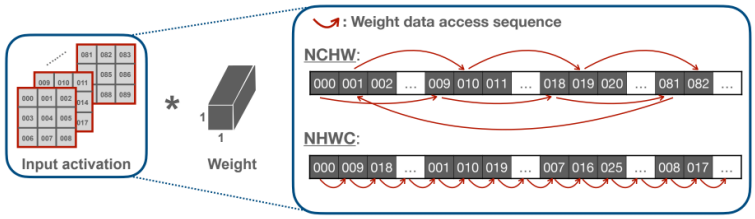
depthwise convolution은 channel별로 연산이 수행되므로, 수행 결과를 기존에 사용하던 memory에 작성하여 peak memory를 줄일 수 있음.



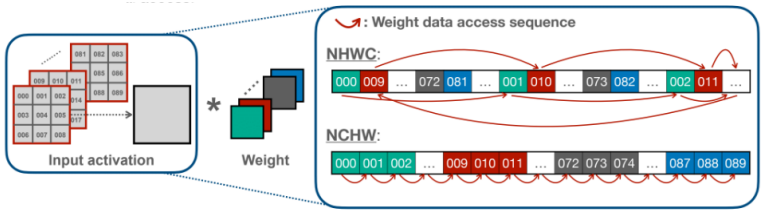
5.4.3. NHWC vs. NCHW

아래와 같이 convolution 연산의 종류에 따라 효율적인 입력 데이터에 대한 memory layout이 다름. TinyEngine은 convolution 연산의 종류에 따라 memory layout을 조정하여 cache miss를 최소화함.

pointwise(1×1) convolution의 경우 매 연산마다 여러 channel에 대해 연산을 수행해야 하므로 아래와 같이 NHWC의 순서대로 데이터를 저장하는 것이 효율적임.

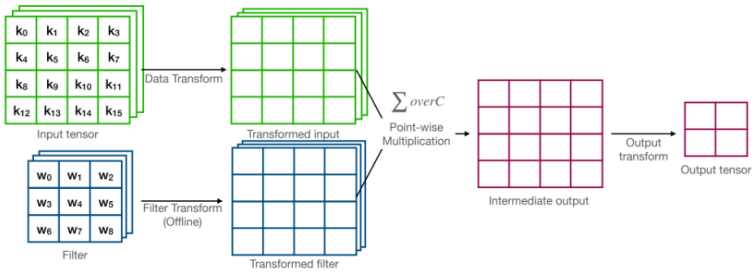


depthwise convolution의 경우 channel 별로 연산을 수행하므로 아래와 같이 NCHW의 순서대로 데이터를 저장하는 것이 효율적임.

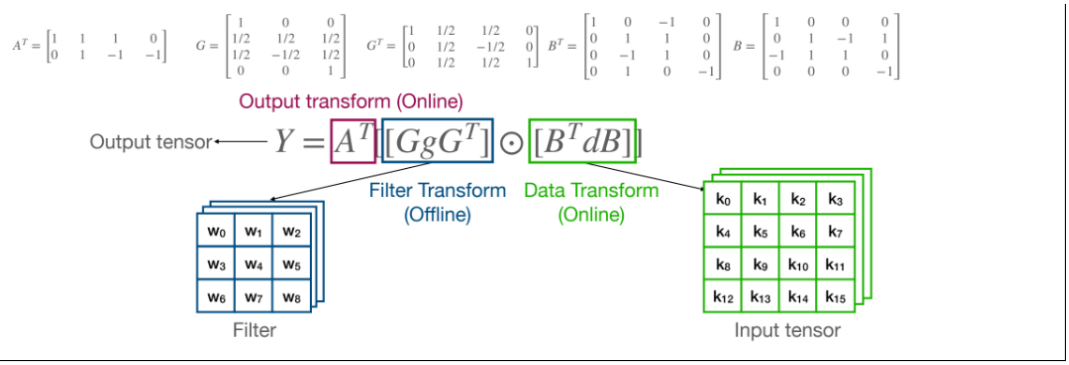


5.4.4. Winograd Convolution

Winograd Convolution은 input activation과 weight를 변환해(transform) 곱셈 연산의 수를 줄이는 기법임.



아래와 같은 수식에 따라 transform을 수행함. 자세한 원리는 논문을 읽어보자. A, G와 관련된 연산은 shift 등으로 구현이 가능하므로 그 값에 따라 곱셈이 필요 없음.



Part III

Domain-Specific Optimization

1. Transformer와 LLM

우선 LLM의 알고리즘을 보고, 이후 가속화를 알아보자. transformer에 대한 자세한 설명은 '인공지능 기초' 필기를 참고하자.

1.1. Transformer Design Variants

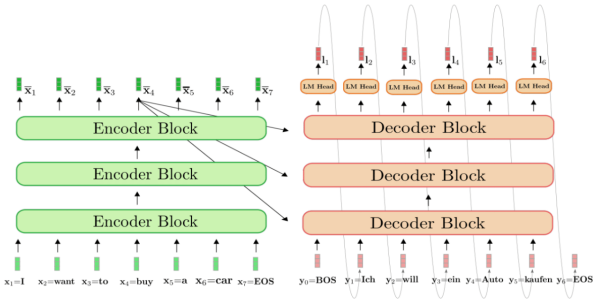
물론 최초에 제시된 형태의 transformer도 자주 사용되었지만, 이를 수정하여 개선한 형태들이 존재함.

1.1.1. encoder/decoder의 활용

transformer의 구조를 일부 수정하여 여러 방식으로 encoder/decoder를 활용할 수 있음.

1. Encoder-deocder(T5)
T5는 구글에서 개발한 NLP model로, encoder-decoder 구조를 가졌음. encoder에서 입력 데이터를 받고, decoder에서 출력 데이터를 생성함.

최초의 transformer는 번역 model이었지만, T5는 단순히 텍스트 입력에 대해 텍스트 응답을 생성함.



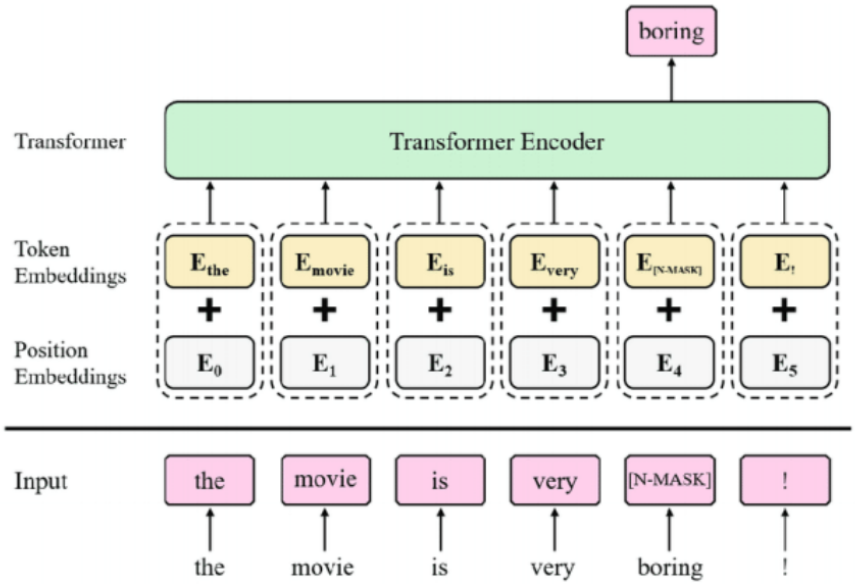
2. Encoder-only(BERT)
BERT(Bidirectional Encoder Representations form Transformers)는 사전 훈련된 NLP model로, encoder만을 활용해서 NLP task를 수행함. BERT는 오늘날 단어에 대한 embedding을 생성할 때 기본적으로 사용되는 모델임.

BERT는 아래와 같이 2가지 종류의 사전 학습 기법을 사용함. 여기에서 NSP는 이제 잘 사용하지 않는다고 함.

1) MLM
MLM(Masked Language Modeling)은 단어에 대한 정확도 학습 기법으로, token의 15%를 MASK token으로 변환한 뒤 해당 token을 예측하는 방식으로 학습함. 빈칸을 뚫어놓고 어떤 단어가 적절한지 맞추는 것.

2) NSP
NSP(Next Sentnece Prediction)은 문장 관계에 대한 정확도 학습 기법으로, sequence의 두 문장이 연속적인지를 판단하는 binary classification을 수행하여 학습함. 이때 입력 sequence의 절반은 실제로 붙어있던 두 문장으로, 나머지 절반은 그렇지 않은 두 문장으로 구성함.

이후 task에 맞춰 fine-tuning한 뒤 사용함.



3. Decoder-only(GPT)
GPT(Generative Per-Trained Transformer)는 사전 훈련된 NLP model로, encoder만을 활용해서 NLP task를 수행함.

GPT는 기본적으로 바로 다음 단어를 예측하는 것을 기본 동작으로 하고, 이에 대해서 사전 학습을 함. 예측 시에는 decoder를 사용하여 앞쪽 token만으로(masking) 단어를 예측함.

GPT 중 작은 model은 fine-tuning을 적용하지만, 큰 model은 fine-tuning을 하는 대신 task를 주거나 예시를 몇 번 보여주면 된다고 함. (궁금하면 더 찾아보자.)

1.1.2. Relative PE

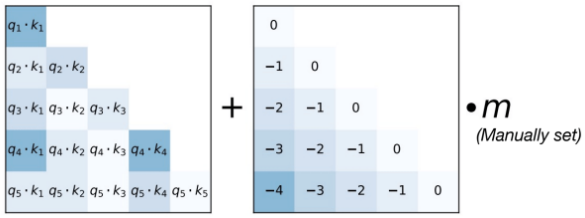
Relative PE는 attention score(Q와 K)를 수정하여 상대적 거리를 나타내는 방식임. 이때 V는 수정하지 않음. 이는 상대적 거리를 사용하여 학습 데이터보다 긴 길이의 데이터도 잘 처리할 수 있도록 하는 데에 목적이 있음.

Absolute PE는 최초의 transformer에서 제안한 PE으로, 위치에 따른 고정된 절대적 값을 embedding(Q와 K, V)에 더하는 방식임. 이 경우 위치 데이터가 transformer 전체에 대해 계속해서 전파됨.

Relative PE에는 ALiBi, RoPE 등이 있음.

2. ALiBi
ALiBi는 아래의 수식과 같이 attention matrix에 Q와 K 사이의 상대적 거리만큼의 offset을 더하는 기법임. 이에 따라 Q와 K에 해당하는 token이 서로 멀수록 반영되는 값이 더 작아져 attention에 덜 반영됨.

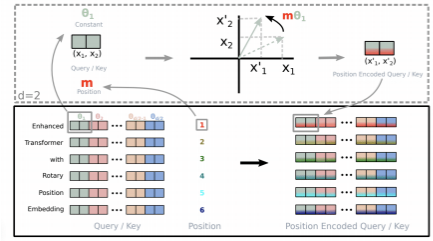
$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}} - m|j-i|)$$



3. RoPE

RoPE는 Q와 K의 각 값을 여러 pair로 쪼갠 뒤, 2차원 공간에서 해당 embedding의 위치만큼을 회전시키는 기법임. ALiBi보다 더 복잡하지만 자주 사용된다고 함(LLaMA 등에서 사용.).

구체적으로는 아래와 같이 각 Q와 K embedding에 대해서 값을 2개씩 묶고, 이 두 값을 좌표로 활용한 벡터를 해당 token의 위치만큼 회전시킴. 즉, 각 embedding의 위치에 따라 rotate하는 정도가 달라짐. 이때 아래 수식과 같이 회전한 Q와 K에 대한 내적은 두 embedding의 상대적인 위치 차이에 따라 결정됨.



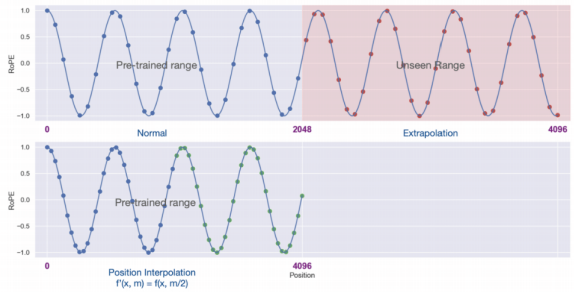
We need a big enough number to distinguish more tokens

$$\Theta = \{\theta_i = \boxed{10000}^{-2(i-1)/d}, i \in [1,2,...,d/2]\}$$

$$\begin{aligned}
\text{RoPE}(x,m) &= xe^{mi\varepsilon} \\
\langle \text{RoPE}(q_j,m), \text{RoPE}(k_j,n)\rangle &= \langle q_j e^{mi\varepsilon}, k_j e^{ni\varepsilon}\rangle \\
&= q_j k_j e^{mi\varepsilon}\overline{e^{ni\varepsilon}} \\
&= q_j k_j e^{(m-n)i\varepsilon} \\
&= \text{RoPE}(q_j k_j, m-n)
\end{aligned}$$

- The phase angle of the inner product of two complex vectors is the phase difference between the two complex vectors (thus m-n)

이때 θ값은 밑(base)을 어떻게 지정하느냐에 따라 크기가 정해짐. θ값이 너무 작으면 여러 token들의 위치 정보가 구분되지 않게 되므로 context length에 따라 base를 충분히 크게 지정하는 것이 중요함 (그렇다고 너무 큰 것도 문제가 된다고 함.). 또한 fine-tuning 시에 더 긴 context를 처리하도록 하기 위해 θ값을 조정하기도 하는데, 이 경우 θ를 조정한 뒤 학습시킨다고 함.



$m \in [0, 2048*2), \quad \theta_i' = \theta_i$ ✗

$m \in [0, 2048*2), \quad \theta_i' = \theta_i/2$ ✓

LLMs usually need to be fine-tuned after extending the context length!

해당 연산을 수식적으로 나타내면 아래와 같음. 행렬 R에 의해 Q와 K가 회전됨.

$$f_{\{q,k\}}(\boldsymbol{x}_m, m) = \boldsymbol{R}^d_{\Theta,m}\boldsymbol{W}_{\{q,k\}}\boldsymbol{x}_m$$

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix}
\cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\
\sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\
0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{\boxed{d/2}} \\
0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2}
\end{pmatrix}$$

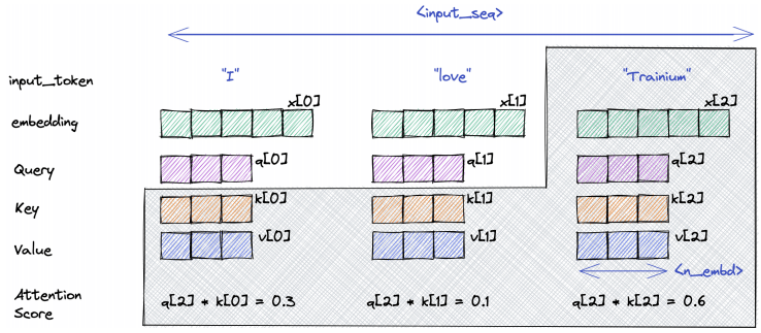
RoPE를 적용하면 적절한 회전 각도를 활용하여 model이 더 긴 문장을 처리할 수 있게 된다고 함.

1.1.3. KV Cache Optimization

1. KV Cache

KV Cache는 attention 연산 시에 K와 V 값을 저장하는 gpu의 memory 공간임.

decoder에서 수행하는 masked multi-head attention에서 특정 token에 대한 attention 연산을 수행할 때는, 자기 자신의 Q와, 활용하는 모든 token들의 K와 V 값을 저장해둬야 함. 이때 attention 연산을 할 때마다 필요한 모든 K와 V를 새로 계산하는 것은 굉장한 비효율이므로, 이전에 계산했던 K와 V를 KV cache에 저장해두고 활용함.



sequence가 길어질수록 필요한 KV cache도 계속 커지므로 너무 많은 memory 공간을 사용하게 될 수 있는데, 이를 MQA 또는 GQA로 개선할 수 있음.

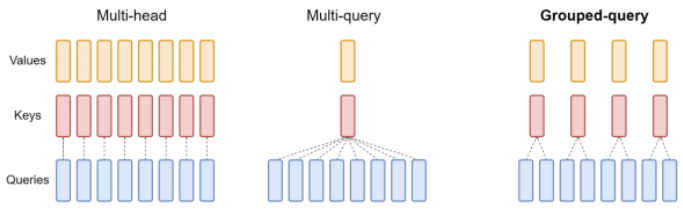
$$\underbrace{BS}_{\text{minibatch}} * \underbrace{32}_{\text{layers}} * \underbrace{32}_{\text{heads}} * \underbrace{128}_{n_{emd}} * \underbrace{N}_{\text{length}} * \underbrace{2}_{\text{K\&V}} * \underbrace{2\text{bytes}}_{\text{fp16}} = 512\text{KB} \times BS \times N$$

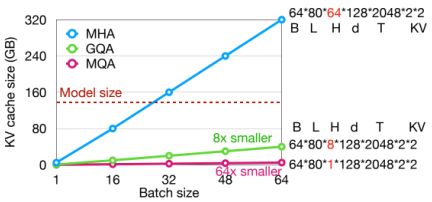
2. MQA/GQA

multi-head attention 대신 MQA 또는 GQA를 사용하여 KV cache의 크기를 줄일 수 있음.

MQA(Multi-query Attention)은 각 head별 Q를 생성하는데, K와 V는 하나만 생성해 공유하는 방식임. MQA는 KC cache의 크기를 크게 줄이지만 표현력이 떨어질 수 있음.

GQA(Grouped-query Attention)은 각 head별 Q를 생성하고, K와 V는 group당 하나씩 생성하여 해당 group의 head들끼리 공유하는 방식임. MQA에 비해 크기는 덜 줄이지만 표현력을 비교적 잘 유지함. 주로 8개의 head를 하나의 group으로 묶는다고 함.



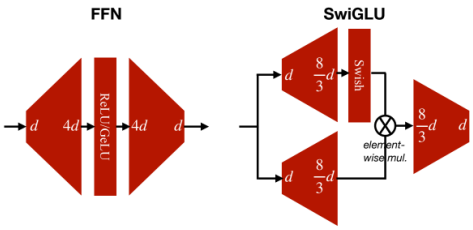


3. KV Cache Quantization

뒤에서 다루는 것처럼 K와 V(KV Cache)에 대해 quantization을 적용하여 memory 사용량을 줄일 수 있음.

1.1.4. GLU

GLU(Gated Linear Unit)는 아래와 같은 구조를 가지는 layer임. transformer에서 FFN대신 GLU를 사용하여 성능을 개선할 수도 있다고 함.



PPL results

Training Steps	65,536	524,288
FFN_{ReLU} (*baseline*)	1.997 (0.005)	1.677
FFN_{GELU}	1.983 (0.005)	1.679
$\text{FFN}_{\text{Swish}}$	1.994 (0.003)	1.683
FFN_{GLU}	1.982 (0.006)	1.663
$\text{FFN}_{\text{Bilinear}}$	1.960 (0.005)	1.648
$\text{FFN}_{\text{GEGLU}}$	**1.942** (0.004)	**1.633**
$\text{FFN}_{\text{SwiGLU}}$	**1.944** (0.010)	**1.636**
$\text{FFN}_{\text{ReGLU}}$	1.953 (0.003)	1.645

1.2. LLM

1.2.1. LLM

LLM(Large Language Model)은 주로 NLP에 사용되는 대규모 언어 model임.

LLM은 model size를 scaling up하는 방향으로 발전하고 있음. 특히 여러 NLP task에 대해 model이 충분히 커야 적정한 성능이 나온다고 함. scaling up 시에는 최적의 학습 computation/accuracy tradeoff를 맞추기 위해 parameter와 학습 token의 개수가 함께 커져야 하고, 이를 Chinchilla Law라고 함. 하지만 inference 측면에서는 작은 model에 대해 많은 학습 시간을 사용해 inference cost를 줄이기도 한다고 함.

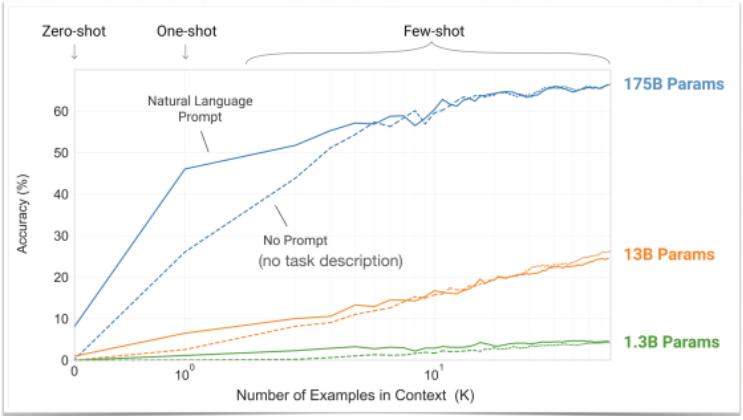
GPT, OPT, LLMA 등 다양한 LLM이 있는데, 각 LLM의 특징을 간단히 살펴보자.

1. GPT-3

GPT-3에서는 기존의 방식처럼 task에 맞춰서 fine-tuning하는 대신, model을 scaling-up하여 zero-shot 및 few-shot으로 fine-tuning 없이 새로운 task를 처리할 수 있도록 했음.

zero-shot은 별도의 fine-tuning이나 예시를 활용하지 않고 바로 새로운 task에 대한 응답을 생성하는 방식이고, few-shot은 별도의 fine-tuning 대신 몇 개의 예시를 활용하여 새로운 task에 대한 응답을 생성하는 방식임.

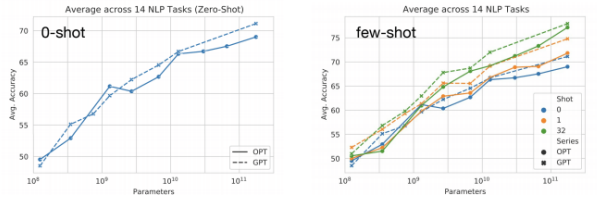
아래 그림과 같이 큰 model일수록, shot이 많아질수록 accuracy가 높아짐.



2. OPT

OPT(Open Pre-trained Transformer Language Model)은 meta에서 개발한 오픈소스 model로, 아래와 같은 스펙을 가졌음.

- Open-source pre-trained LLM from Meta
- Size: 125M/350M/1.3B/2.7B/6.7B/13B/30B/66B/175B
- Design choices: Decoder-only, Pre-norm (post-norm for 350M), ReLU activation in FFN
- 175B model: hidden dimension 12288, #heads 96, vocab size 50k, context length 2048
- Close performance compared to GPT models



3. LLaMA

LLaMA(Large Language Model Meta AI)는 meta에서 개발한 오픈소스 model로, OPT보다 efficiency를 더 고려했음.

LLaMA는 SwiGLU, RoPE 등을, Llama 2는 GQA 등을 활용하고, Llama 3는 training token 수를 대폭 늘리는 등 여러 기법이 적용되어 있음.

참고로 NLP의 task는 아래와 같이 2가지로 분류할 수 있음.

1) Discriminavtive task : 입력 데이터에 대한 판별을 수행함.
2) Generative task : 입력 데이터를 활용해 데이터를 생성함.

1.2.2. Perplexity

Perplexity(PPL)는 LLM의 성능을 나타내는 지표로, LLM이 예측을 수행할 때의 불확실성을 나타냄. 즉, 이 값이 낮을수록 성능이 좋은 model임.

아래와 같은 수식에 의해 계산됨. 이때 N은 token의 개수이고, $P(t_i|t_{1:i-1})$는 이전 token을 기반으로 예측한 현재 token에 대한 확률임(확률 계산 방법은 model에 따라 달라질 수 있음.). 각 token에 대한 확률이 높다는 것은 model의 예측이 명확하다는 것이고, 확률이 낮다는 것은 model의 예측이 불확실하다는 것임. perplexity는 이를 나타내는 지표임.

$$Perplexity = (\frac{1}{\Pi_{i=1}^{N} P(w_i|w_{1:i-1})})^{\frac{1}{N}}$$

2. LLM Deployment Techniques

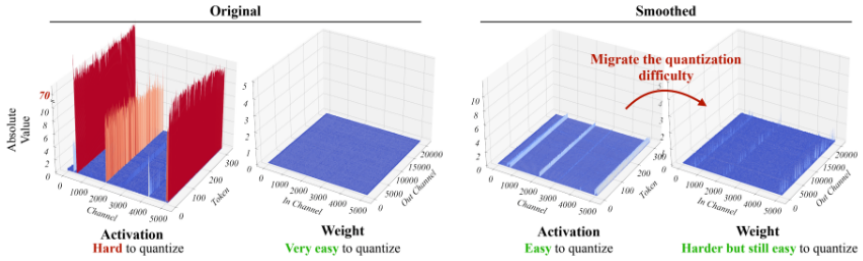
이제 LLM을 최적화하는 방법을 알아보자.

2.1. Quantization 적용

2.1.1. Weight-Activation Quantization : SmoothQuant

SmoothQuant는 activation을 특정 값으로 나누고 weight는 동일한 값을 곱해, LLM에 대한 quantization(W4A4 quantization. INT4와 INT4) 시에 발생하는 activation에서의 degradation을 줄이는 기법임. 즉, quantization 난이도를 activation에서 weight로 옮김. 이는 주로 cloud에서 사용하는 기법임.

이전에 다룬 quantization 기법들을 LLM에 naive하게 적용하면 model size가 커질수록 accuracy가 떨어짐. 이는 model size가 커지면 activation에서 큰 outlier(이상치)가 발생하기 때문임. outlier 값이 크면 평범한 값들은 0으로 quantize되거나, dynamic range가 훼손됨. 이에 따라 smoothquant에서는 activation 값을 작게 하고, 그만큼 weight 값을 키워 출력값은 유지하면서 quantization을 적용하기 쉽도록 model을 변형함. 이때 activation에서 outlier가 발생하는 지점(channel)은 고정된다고 함.

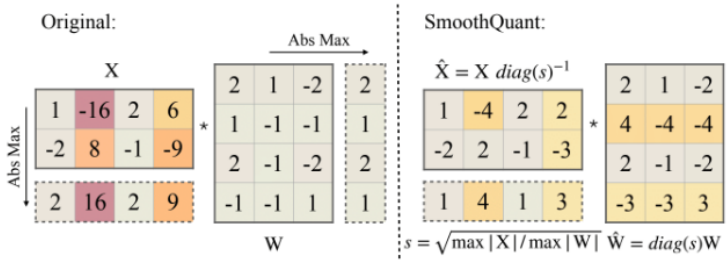


SmoothQuant에서는 아래와 같이 s_j를 계산한 뒤(Calibration Stage), activation에는 행에 대해 element-wise로 나누고, weight에는 열에 대해 element-wise로 곱함(Smoothing Stage). 이후 inference 시에는 smoothing된 model에 quantization을 적용하여 활용할 수 있음.

이때 α는 Migration Strength로, activation에서 weight로 전가하는 값의 크기를 지정함. α가 너무 작으면 activation이 여전히 quantize하기 어렵고, 너무 크면 weight가 quantize하기 어려워지므로 적정한 값을 지정하는 것이 중요함. 대체로 0.5가 sweet spot이라고 함.

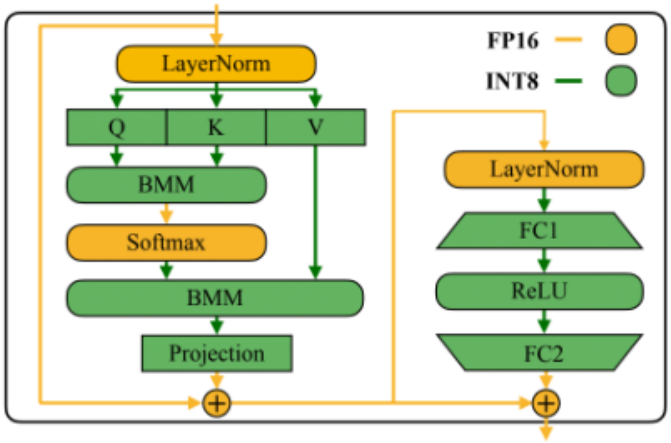
$$s_j = \frac{\max(|X_j|)^{\alpha}}{\max(|W_j|)^{1-\alpha}}$$

$$Y = (Xdiag(s)^{-1}) \cdot (diag(s)W) = \hat{X}\hat{W}$$



LLM에 SmoothQuant를 적용하면 quantization을 적절히 적용할 수 있게 되고, 이에 따라 quantize하기 전에 비해 inference 속도가 빨라지고 memory 사용량이 적어짐. 이때 fine-tuning 없이도 accuracy을 잘 보존한다고 함. 또한 하드웨어 사용 측면에서도, latency를 유지하면서 더 적은 gpu를 사용할 수 있는데 gpu를 덜 사용하면 gpu 간 communication이 적어지므로 이에 따른 overhead가 적고 병렬 처리가 편리해짐.

아래와 같이 SmoothQuant를 적용하여 transformer에 quantization을 적용할 수 있음.



LLM 중 SwishGLU를 사용하는 LLaMA는 quantization을 적용하기 더 어려운데, SmoothQuant을 쓰면

적용이 가능하다고 함.

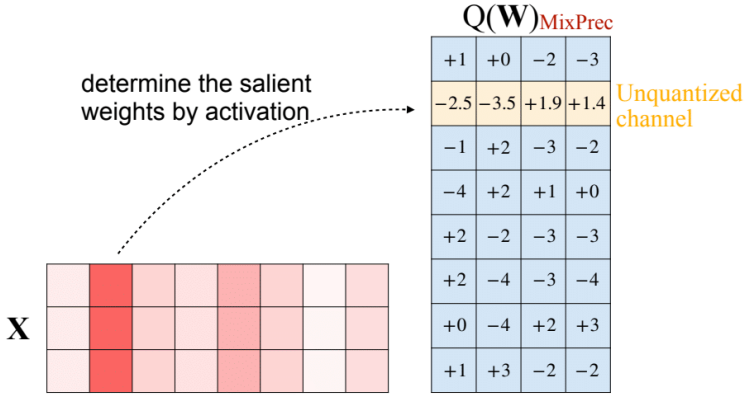
2.1.2. Weight-Only Quantization : AWQ/TinyChat

1. AWQ

AWQ(Activation-aware Weight Quantization)란 activation을 고려하여 weight에서 1% 가량의 Salient(중요한) weight를 찾고, 이를 scale up한 뒤 quantization(해당 논문에서는 W4A16 weight-only quantization, INT4와 FP16.)을 적용하는 방식임. 이는 주로 edge에서 사용하는 기법임.

single batch를 주로 처리하는 edge ai 등에서의 model에 대해서는 activation이 아니라 weight가 병목임. 이 경우 weight가 activation에 비해 훨씬 많은 memory 공간과 latency를 가지므로, weight only quantization을 적용함. 이때 naive하게 기존의 기법들을 사용하면 perplexity 측면에서 degrade가 꽤 큰데, AWQ를 적용하여 적절히 quantizatoin할 수 있음.

AWQ의 핵심 아이디어는, weight의 모든 부분이 중요한 것은 아니므로, quantization 시에 1퍼센트의 salient(중요한) weight를 quantize하지 않으면 quantization 이후에도 성능이 유지된다는 것임. 문제는 어떻게 salient weight를 찾을 것인가인데, AWQ에서는 weight 대신 activation으로 salient weight를 찾음. 앞에서 본 것처럼 activation은 outlier를 가지고, 그렇게 값이 큰 부분과 곱해지는 weight 부분은 전체 결과에 더 큰 영향을 주므로 salient하다고 할 수 있음(실제로 이런 방식의 성능이 더 좋다고 함).



이때 salient weight를 quantize하지 않는다면 정수와 실수를 동시에 저장해야 하는데, 이는 하드웨어적으로 비효율적임. 이에 따라 AWQ에서는 SmoothQUant에서처럼 값 s를 weight에 곱하고, activation에 나눠 salient weight를 scale up함. 이때 s는 salient weight($w \in W$)의 값만 scale up하고, 이후 quantization을 적용함. 즉, 아래와 같이 quantize함.

$$Q(w) = \Delta \cdot round(\frac{w}{\Delta}), \quad \Delta = \frac{\max(|w|)}{2^{n-1}}$$

$$WX \rightarrow Q(W \cdot diag(s))(diag(s)^{-1} \cdot X) \quad or \quad Q(w \cdot s) \cdot \frac{x}{s}$$

2. 값 s 찾는 방법

실제로 아래와 같이 round에서 발생하는 오차가 s로 나누어지므로 그 값이 작아져 quantization에 의한 degrade가 줄어듦.

$$Err(Q(x) \cdot x) = \Delta \cdot Err(Round(\frac{w}{\Delta})) \cdot x \Longrightarrow Err(Q(x \cdot s) \cdot \frac{x}{s}) = \Delta \cdot Err(Round(\frac{w \cdot s}{\Delta'})) \cdot x \cdot \frac{1}{s}$$

이때 s값이 너무 커지면 Δ가 크게 달라져 성능이 다시 떨어질 수 있음. 즉, s가 너무 작으면 salient가 소실되고, 너무 크면 non-salient가 소실됨. 적절한 s값은 아래와 같은 cost function이 최소가 되도록 하는 α를 찾아 얻을 수 있음. 여기에서 s_X는 activation이 가지는 값들의 크기의 평균, α는 0부터 1까지의 값을 가지는 hyperparameter임.

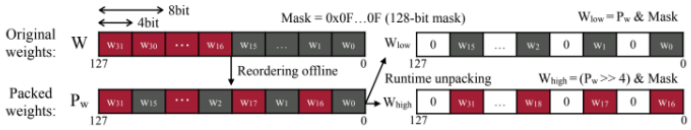
$$L(s) = ||Q(W \cdot s)(s^{-1} \cdot X) - WX||$$

$$s = s_X^{\alpha}, \ \ \alpha^* = argmin_{\alpha} L(s_X^{\alpha})$$

3. TinyChat
TinyChat은 edge에서 돌아가는 LLM inference engine으로, AWQ를 구현하는 데에 사용될 수 있음. TinyChat은 efficient하고, 가볍고, 파이썬 기반이면서 여러 platform에서 사용이 가능하다는 이점이 있음.

아래와 같은 기법으로 LLM inference를 efficient하게 수행함.

1) Hardware-aware Packing
컴퓨터는 기본적으로 데이터가 byte 단위로 저장되는데에 반해, W4A16에서는 각 weight를 INT4로 저장함. 이에 따라 TinyChat은 하나의 byte에 2개의 weight값을 저장하는데, decode하기 쉽도록 순서대로 저장하는 대신 아래 그림과 같이 번갈아 가며 저장(packing)함. decode 시에는 mask와 shift로 쉽게 값을 얻을 수 있음.



2) Kernel Fusion
앞에서 다룬 것처럼, kernel launch overhead 등에 의해 gpu에서의 kernel call에는 overhead가 존재하므로 최대한 한 번의 kernel call에 여러 연산을 fuse하는 것이 좋음. 이에 따라 attention 연산 및 dequantization 등의 작업 각각을 fusion 처리하여 한 번의 kernel call로 연산함.

직관적으로도 scale up 하면 quantization 이후에도 해당 값을 구분할 수 있게 됨. 예를 들어, 0.5과 1 모두 1로 quantize되는 경우 2를 곱하면 1과 2로 quantize됨.

$argmin_x f(x)$는 f(x)를 최소화하는 x를 찾는 연산을 의미함.

AWQ는 LLM, visual-language multimodal 등에 적용되어 단순한 구현으로 efficiency를 확보할 수 있도록 함.

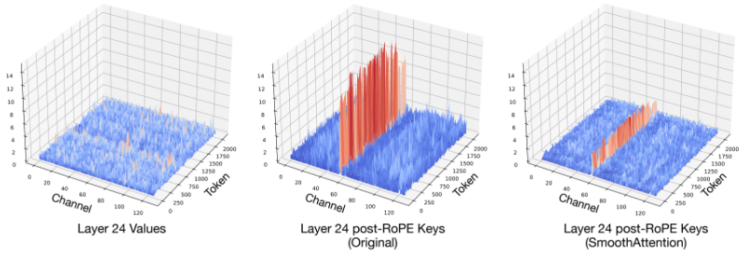
2.1.3. QServe

QServe(W4A8KV4)는 cloud에서의 W8A8KV8(INT8)(SmoothQUant)과, edge에서의 W4A16KV16(INT4, FP16)(AWQ) 각각의 장점을 활용한 기법임.

QServe에서는 아래와 같은 기법들을 사용함.

1) SmoothAttention
SmoothAttention은 SmoothQuant의 아이디어를 KV cache에 적용한 기법임.

아래 그림과 같이 V는 대체로 smooth하지만, K에는 outlier가 존재할 수 있어 기존의 quantization을 단순 적용하면 4bit로 quantize하기 어려움. 이에 따라 Q와 K가 곱해지므로 K에서 Q로 난이도를 옮겨 quantize할 수 있음.



$$\mathbf{Z} = (\mathbf{Q}\mathbf{\Lambda}) \cdot (\mathbf{K}\mathbf{\Lambda}^{-1})^T, \quad \mathbf{\Lambda} = \text{diag}(\lambda)$$

$$\lambda_i = \max(|\mathbf{K}_i|)^{\alpha}$$

2) dequantize 시에 곱셈 연산을 먼저 수행
Z, S를 활용한 dequantize 시에 곱셈 연산을 먼저 수행하여 overflow가 발생하지 않도록 함.

구체적인 내용은 논문을 읽어보자.

참고로 W4A4 등의 quantization 방식은 memory는 많이 아끼지만, accuracy loss가 심하고 GPU에서 효율적으로 처리되지 못함. 특히 CUDAcore와 Tensorcore를 적절히 활용하지 못함.

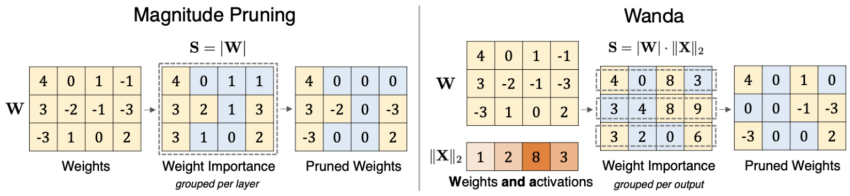
2.2. Pruning 적용

nature language에는 redundancy가 많으므로 이를 적절히 pruning할 수 있음.

2.2.1. Weight Sparsity : Wanda

Wanda는 activation을 활용하여 weight를 pruning하는 방식임.

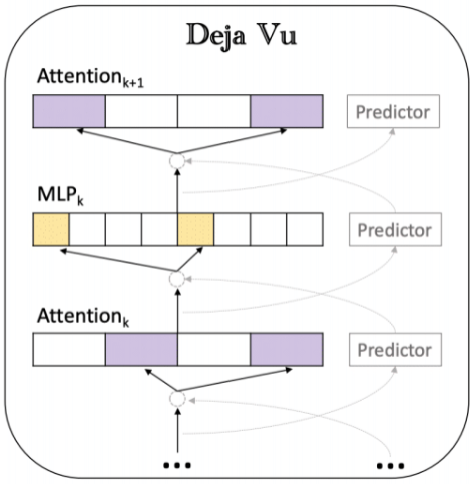
기존의 naive한 방법으로는 weight의 크기에 따라 weight를 pruning했지만, Wanda에서는 AWQ와 유사하게 weight가 아니라 activation을 기준으로 weight를 pruning함. 즉, 아래와 같이 입력 activation 값과 W의 절댓값을 element-wise하게 곱한 뒤, 그 결과가 작은 위치의 weight를 pruning함. 이에 따라 activation 결과에 영향을 덜 미치는 것이 제거됨.



2.2.2. Contextual Sparsity : DejaVu, MoE

1. DejaVu
DejaVu는 predictor를 활용하여 contextual sparsity를 구현하는 방식임.

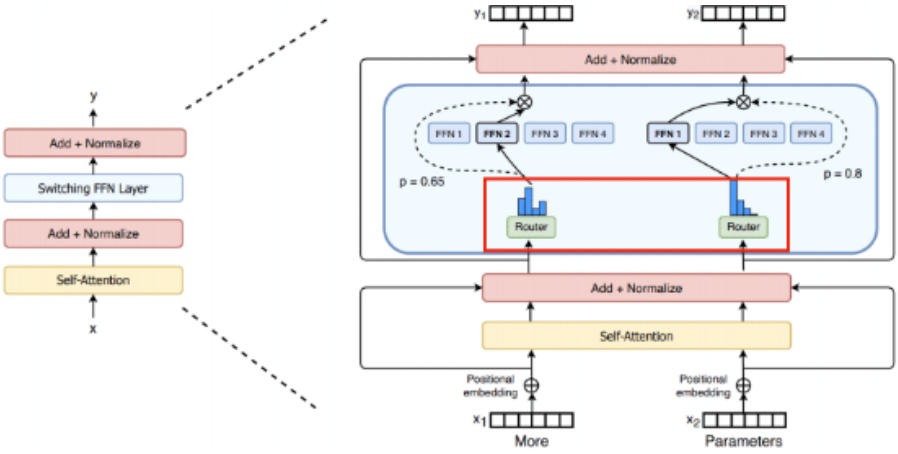
activation에 대한 pruning은 입력에 종속적이므로 static하게 처리하는 것은 accuracy를 훼손할 수 있음. 반면 Contextual Sparsity는 입력에 독립적으로 존재하는 head나 feature의 작은 부분에 대한 sparsity임. DejaVu에서는 아래와 같이 predictor를 사용하여 contextual sparsity의 발생을 예측함. 자세한 내용은 논문을 읽어보자.



2. MoE

MoE(Mixture-of-Experts)는 Experts라고 부르는 FNN을 여러 개 사용하고, Router로 token을 각 experts로 분배하는 방식임. 이에 따라 course grained sparsity를 구현함.

router는 아래와 같이 각 token이 서로 다른 weight를 사용할 수 있도록 분배함. 이에 따라 model size를 크게 하면서도 각 inference에서의 overhead가 커지지 않도록 할 수 있음. 이때 routing하는 기법에도 여러 정책이 존재함.



여기서 각 expert가 한 번에 처리할 수 있는 token의 개수를 Capacity라고 하고, 이는 아래와 같이 계산할 수 있음. 즉, Capacity Factor에 따라 capacity가 지정됨.

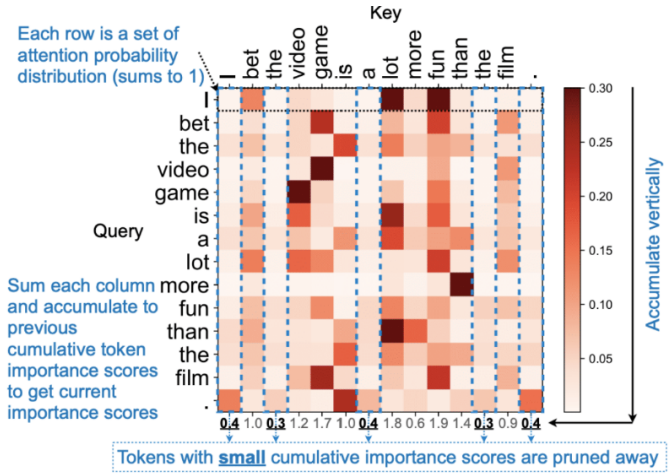
$$Capacity = \frac{tokens\ per\ batch}{number\ of\ experts} \times Capacity\ Factor$$

2.2.3. Attention Sparsity : SpAtten

1. SpAtten

SpAtten은 attention score 등을 활용하여 token 또는 head를 pruning하는 기법임.

아래 그림과 같이 attention map에서 token 별 attention score(Q와 K의 곱) 합계를 구하고, 합계가 작은 token은 pruning함. 또한 해당 token에 대해서는 V를 fetch하지 않도록 함.



2.3. LLM 지원 시스템

2.3.1. Metrics for LLM

LLM에 대한 metric들로는 아래와 같은 것들이 있음.

LLM의 efficiency 측면에서의 목표는 TTFT/TPOT를 줄이고, throughput을 늘리는 것임. 이때 TPOT와 throughput은 tradeoff 관계에 있음. 여러 request를 한 번에 처리하면 throughput은 늘어나지만 TPOT는 줄어듦.

1. TTFT
TTFT(Time To First Token)는 사용자의 입력으로부터 첫 번째 token이 생성되기까지 걸리는 시간임. prompt processing 시간과 실질적인 token 생성 시간에 따라 결정됨. 특히 real-time으로 동작하는 model에 대해 중요함.

2. TPOT
TPOT(Time Per Output Token)는 시간당 생성하는 토큰의 수임. 사용자가 model의 속도를 어떻게 느끼는가를 결정함. 대체로 10 tokens/second를 보다 더 걸리면 느리다고 느끼고, 덜 걸리면 빠르다고 느낀다고 함.

3. Latency
이에 따라 latency는 아래와 같이 계산됨.

$$Latency = TTFT + (TPOT \times 생성될\ token의\ 수)$$

4. Throughput
Throughput은 inference server가 여러 request에 대해 시간당 생성하는 토큰의 수임.

5. 주요 Heuristics
LLM의 성능 평가와 관련해서는 아래와 같은 heuristic들이 있음.

1) output length는 latency에 영향이 큼.

2) input length는 성능 면에서는 영향이 크지 않지만, 하드웨어 측면에서는 영향이 큼. input length는 TTFT 등에 영향을 줄 수 있음.

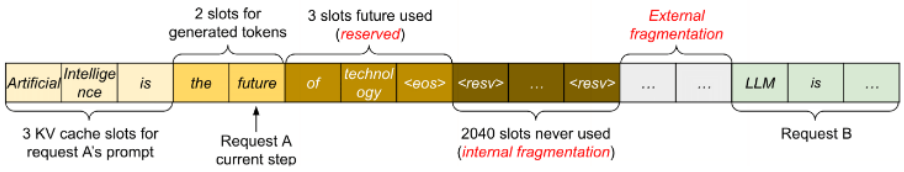
3) model size가 클수록 latency가 커짐. 단, 정비례하지는 않음.

2.3.2. Paged Attention

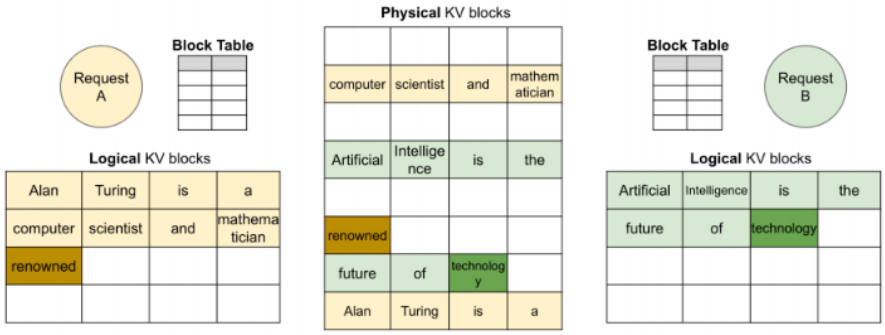
Paged Attention은 KV cache에 paging 기법을 적용하여 관리하는 방식임.

KV cache는 GPU의 내장 memory에 위치하고, 이에 따라 internal/external fragmentation, reservation

등에 의한 memory 낭비가 발생할 수 있음.



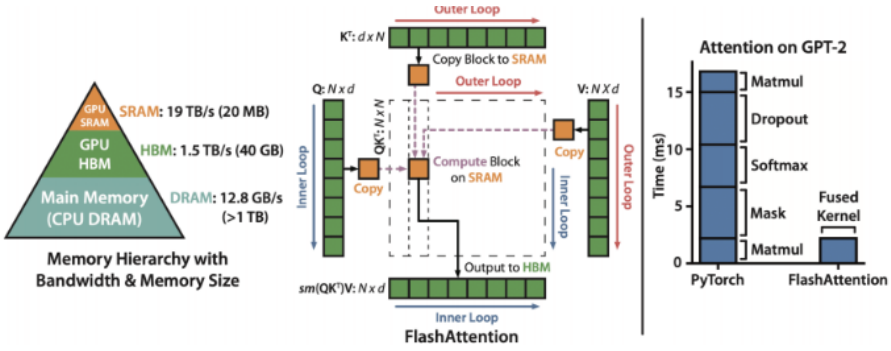
이에 따라 KV cache에 대해 paging을 적용하여 논리적으로는 연속적인 공간을 사용하면서, 물리적으로는 비연속적이고 효율적으로 데이터를 저장할 수 있음. 또한 여러 request를 병렬 처리하거나 여러 개의 응답을 동시에 생성하는 등의 작업을 더 잘 수행할 수 있음.



2.3.3. FlashAttention

FlashAttention은 attention matrix의 모든 부분을 연산하는 대신, tiling하여 SRAM에 올려 연산하도록 한 방식임.

GPU 관점에서 memory 구조는 아래와 같으므로 성능이 향상된다고 함.



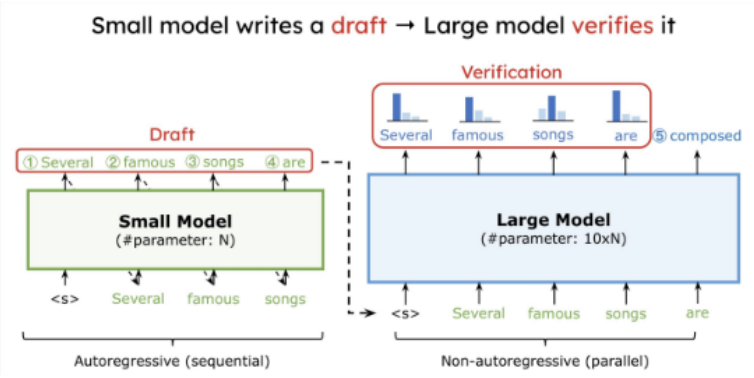
2.3.4. Speculative Decoding

Speculative Decoding은 autoregressive model의 순차적인 처리에 의한 병목을 해결하기 위해 draft model로 generation을, target model로 verification을 수행하도록 하는 기법임. 이를 통해 infererence를 가속화함.

LLM의 generation은 token by token으로 처리되고 attention 연산을 수행해야 하므로, 대체로 memory가 병목임.

speculative decoding에서는 2개의 model을 사용함. 하나는 Draft Model로, generation을 위한 작은 모델임. 다른 하나는 Target Model로, verification을 위한 큰 모델임. speculative decoding은 draft model이 autoregressive하게 k개의 token을 생성하면, target model이 k개의 token을 병렬적으로 처리하여 각 token에 대한 예측값을 도출함. 예측값이 실제 값과 일치하면 token을 사용하고, 틀리면 draft model에서 token을 다시 생성함.

draft model로 generate하므로 memory를 아낄 수 있고, target model은 병렬 처리를 하므로 빠른 작업이 가능함.



Autoregressive는 이전에 생성된 token들이 다음 단계의 입력으로 사용되는 것을 말함.

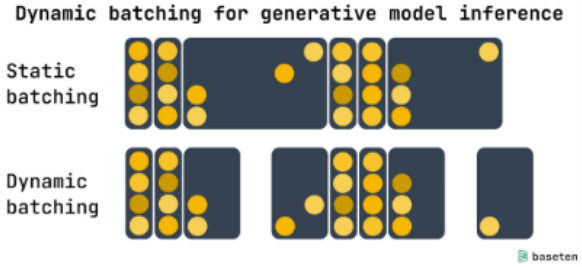
2.3.5. Batching

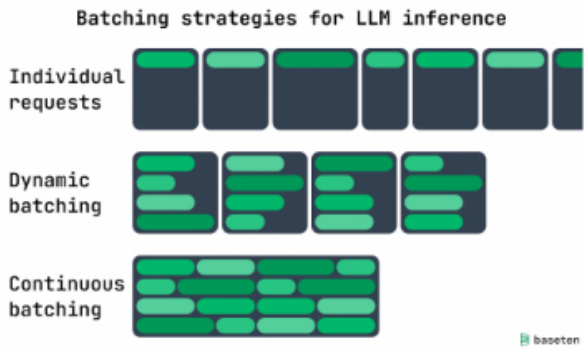
Batching을 통해 CPU 및 GPU 자원을 효율적으로 활용할 수 있음. LLM에서의 batching에는 아래와 같은 기법들이 존재함.

1) Static Batching : batch가 꽉 찰 때까지 기다렸다가 꽉 차는 순간 처리하는 방식임. offline task(데이터를 한 번에 모아서 처리)에 대해서는 효과적이지만, online task에 대해서는 latency가 발생할 수 있음.

2) Dynamic Batching : batch가 꽉 차거나 일정 시간이 지나면 처리하는 방식임.

3) Continuous Batching : token-by-token으로 처리하는 방식임. 즉, 특정 token에 대한 처리가 끝나면 바로 다음 token에 대한 처리를 시작함.





3. LLM Post-Training

LLM의 학습은 pre-training data collection, pre-training, post-training, optimization의 순서로 수행되는데, 여기에서는 그 중 post-training에 대해 살펴봄.

3.1. LLM Fine-tuning

3.1.1. SFT

SFT(Supervised Fine-tuning)은 말 그대로 입력과 출력(ground truth)의 쌍을 활용하여 LLM을 fine-tuning하는 것임.

이를 통해 학습된 model을 특정 도메인이나 task에 최적화하고, 사용자의 preference(helpfulness, safty 등)에 맞출 수 있음.

각 token에 대해서 loss는 아래와 같이 계산될 수 있음. 즉, $u_0, \cdots, u_{i-1}$에 대해서 u_i가 등장할 확률에 로그를 취한 값을 더하는데, 이 값이 클수록 잘 예측하는 것.

$$L(U) = \sum_i \log P(u_i | u_0, \cdots, u_{i-1}; \Theta), \;\; Loss = -L(U)$$

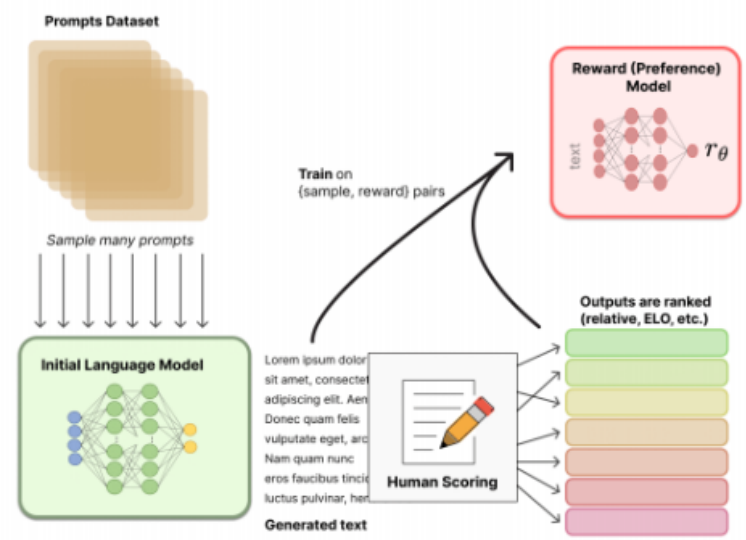
3.1.2. RLHF/DPO

1. RLHF
RLHF(Reinforcement Learning from Human Feedback)는 사람의 판단에 따라 model의 출력 결과를 분류하여 reinforcement learning에 활용하는 방식임.

이를 활용하면 SFT와 마찬가지로 creativity, truthfulness, usefulness 등 단순 학습만을 적용한 model이 가지기 어려운 특성을 반영할 수 있음. 즉, LLM을 사용자의 preference에 맞출 수 있음.

RLHF는 아래의 과정을 따라 수행됨.

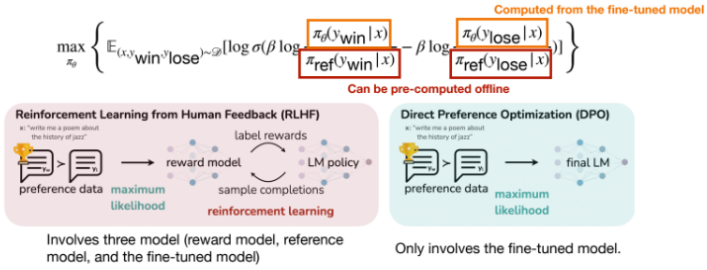
1) 학습된 model에 SFT를 적용함.
2) model을 활용하여 하나의 입력에 대한 출력을 여러 개 생성하고, 사람(labeler)이 각 출력에 등급을 매김.
3) 등급이 매겨진 데이터를 활용하여 reward model을 학습시킴.
4) reward model을 활용하여 model을 강화학습함.



2. DPO

DPO(Direct Preference Optimization)은 RLHF를 SFT task로 단순화한 기법임. RLHF에서는 reward model과 model에 대해 각각 학습을 수행해야 했는데, DPO에서는 이를 한 번의 학습으로 처리함.

DPO는 아래와 같은 수식의 값을 최대화하는 policy model π_θ를 찾는 과정임. reference model은 parameter가 고정된 model로, 이를 활용하여 policy model을 학습시킴. 이때 refernece model에 대한 연산은 pre-compute될 수 있으므로 이 수식은 단순 SFT task로 볼 수 있음.



이와 같이 사용자의 preference에 대해 alignment를 수행하는 기법들은 특히 ChatGPT같은 시스템의 개선에서 중요함.

3.1.3. PEFT

PEFT(Parameter Efficient Fine-tuning)는 fine-tuning을 적용할 parameter를 최소화하는 등의 처리를 통해, LLM fine-tuning에 대한 efficiency를 확보하는 기법임.

1. BitFit

BitFit은 bias 또는 일부 bias에 대해서만 fine-tuning을 적용하는 기법임. 즉, sparse하게 fine-tuning함.

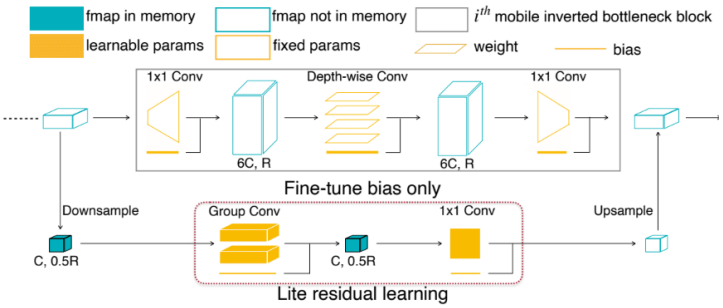
weight에 비해 bias는 그 수가 확연히 적으므로 fine-tuning 비용을 줄일 수 있음.

작거나 중간 크기의 model에 대해서는 전체 parameter를 fine-tuning하는 것과 유사한 성능을 보이지만, 큰 크기의 model에 대해서는 성능 저하가 발생한다고 함.

2. TinyTL

TinyTL에서는 Lite Residual Learning을 적용함.

즉, 아래와 같이 크기가 큰 main branch(윗 부분)는 학습하지 않고, 크기가 작은 side(residual) branch(아랫 부분)에 대해서만 학습을 수행함. 이때 activation의 크기를 작게 유지해 비용을 낮춤.

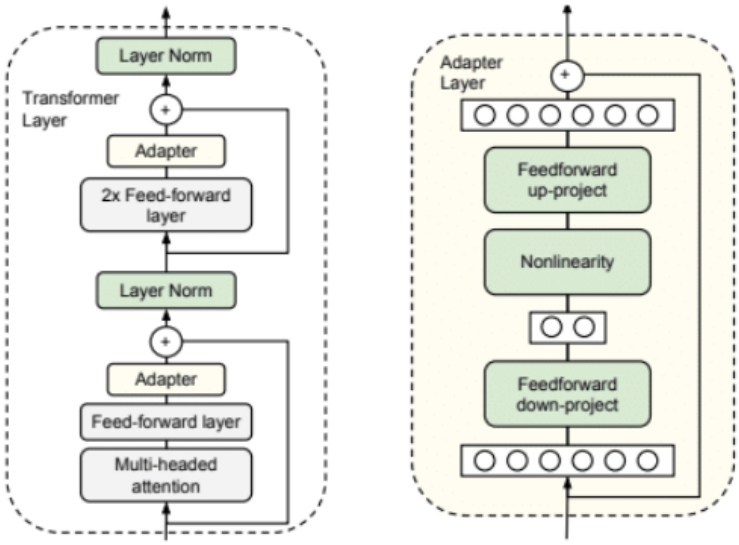


3. Adapter

Adapter는 transformer 아키텍처 중간에 adapter layer를 추가로 삽입하는 기법임.

Adapter Layer는 아래와 같이 bottleneck 구조의 layer로, 특정 task별로 존재함. fine-tuning 시에는 각 task에 대해 model 전체에 대해 학습하는 대신 adapter layer만 학습시킴. 이렇게만 해도 거의 state-of-art만큼의 성능을 낸다고 함.

물론 adapter layer를 추가하면 model이 더 깊어지는 것이므로 inference에서 추가적인 latency가 발생하는 overhead가 있음. prompt-tuning 또는 prefix-tuning을 사용하면 추가적인 parameter를 사용하지 않을 수 있음.

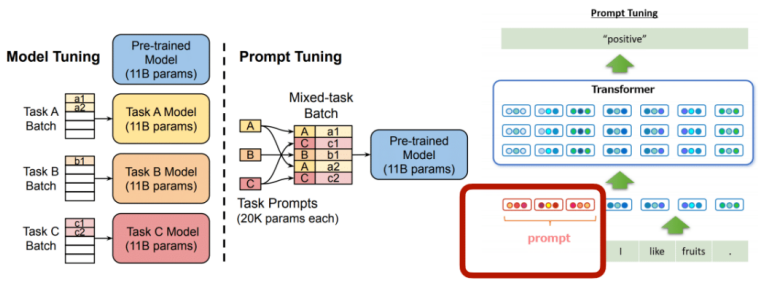


4. Prompt-Tuning

Prompt-Tuning은 입력 prompt에 learnable prompt(token)를 추가하는 기법임. 이때 추가하는 prompt는 task별로 존재함. 즉, 첫 번째 layer의 입력 prompt를 tuning함.

prompt-tuning에서는 추가적인 parameter를 사용하지 않으며, 기존 parameter는 수정되지 않음. 즉, 하나의 pre-trained model만을 사용해서 각 입력에 대한 task를 처리함.

model이 충분히 크면 단순 fine-tuning만큼의 성능이 나온다고 함.

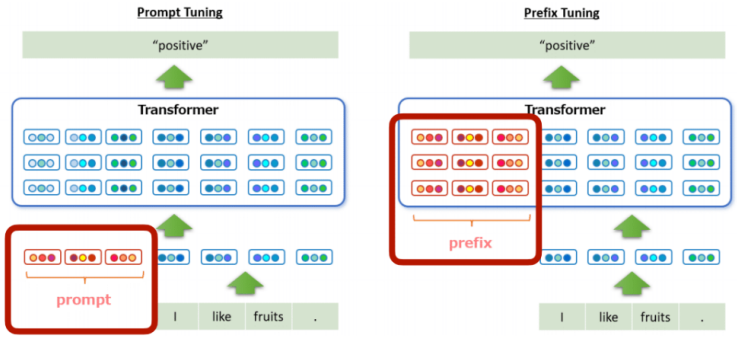


5. Prefix-Tuning

Prefix-Tuning은 모든 layer에서의 입력에 learnable prompt를 추가하는 기법임.

prompt-tuning보다 성능이 좀 더 좋다고 함.

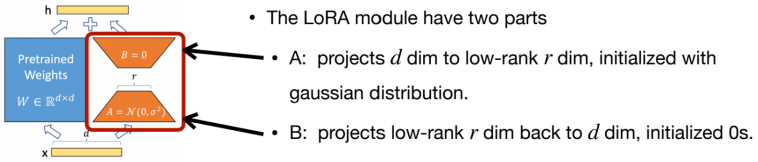
prompt-tuning과 prefix-tuning 모두 prompt 추가하므로 input length가 길어지는데, 이에 따라 KV cache size가 커지고 inference latency가 길어짐. 또한 실제로 처리 가능한 length가 줄어들 수도 있음. LoRA 등을 활용하면 추가적인 inference latency 없이 fine-tuning할 수 있음.



6. LoRA

LoRA(Low Rank Adaptation)는 TinyTL에서와 유사하게, 각 layer에 대해 trainable rank decomposition matrix(adapter)를 side branch로 사용하는 기법임. fine-tuning 시에 original branch는 학습시키지 않고, side branch만을 학습시킴.

rank decomposition matrix는 아래와 같이 두 부분으로 나뉨.



즉, 출력은 아래와 같이 계산되고, 학습이 완료된 이후 A와 B를 fuse할 수 있음.

$$h = xW + xAB = x(W + AB) = xW'$$

LoRA를 적용하는 것과 더불어, base(pre-trained) model에 대한 quantization 또한 적용하여 memory를 더 줄이는 QLoRA라는 기법도 있음.

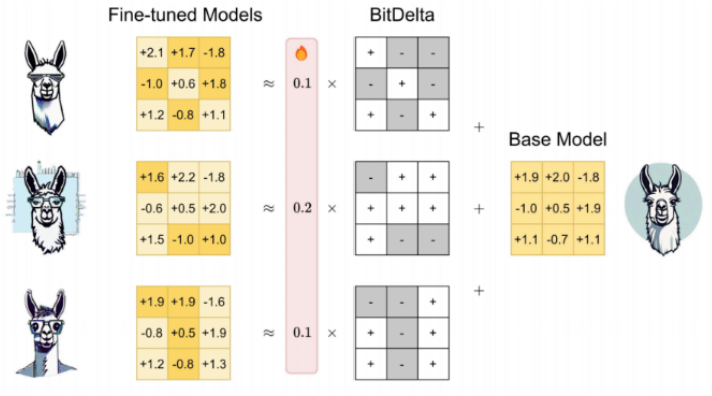
7. Bit-Delta

Bit-Delta는 fine-tuning이 적용된 model을 비트 수준으로 압축하는 기법임.

fine-tuning은 기존 pre-trained model에 적은 정보만을 더하므로, 이와 같이 redundancy를 고려하여 정보를 더욱 압축할 수 있음. 이에 따라 memory를 아끼고 latency를 줄일 수 있음.

Bit-Delta에서는 아래와 같이 fine-tuning이 적용된 model과 base model 사이의 weight에 대한 delta(변화량)를 1비트로 quantize(sign function 적용)하고, 해당 delta에 곱해지는 per tensor scaling factor(α)를 fine-tuning하는 기법임. 이때 α는 해당 tensor가 가지는 값들의 크기의 평균을 초기값으로 가지고, quantization error를 최소화하도록 학습됨.

$$\Delta = W_{fine} - W_{base}, \quad \hat{\Delta} = \alpha \cdot Sign(\Delta)$$



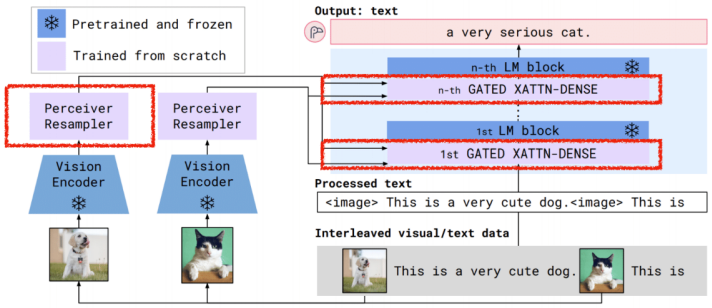
3.2. Multi-modal LLM

multi-modal(visual-language) LLM을 처리하는 방법으로는 cross-attention을 이용하는 것(Flamingo style)과 visual token을 단순 입력으로 보는 것(PaLM-E style)이 있음.

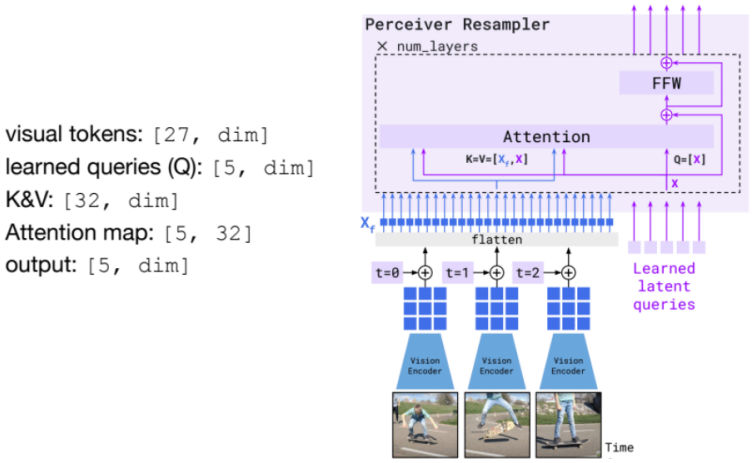
3.2.1. Cross-Attention Based : Flamingo

Flamingo는 deepmind에서 개발한 visual language model로, cross-attention 연산을 통해 multi-modal 처리를 함.

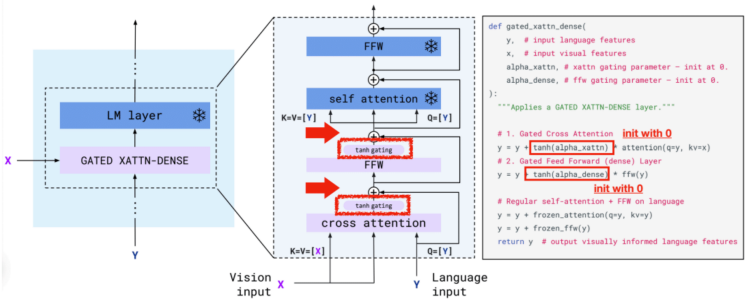
Flamingo에서 cross-attention 연산은 아래와 같이 수행됨. 입력에서 (image를 제거한) lanuage 부분은 LLM에 넣고, image 부분은 vision encoder에 넣어 token화한 뒤 perceiver resampler를 거쳐 LLM의 K와 V로 사용됨.



perceiver resampler를 포함하는 image를 처리하는 부분은 아래와 같은 구조임. 이때 입력으로 들어오는 image의 총 길이는 가변적이지만, learned query의 길이는 고정되어 있고 이에 따라 출력 길이가 결정됨(attention 연산을 생각하면 당연함.). 이때 learned query는 learnable query vector임.



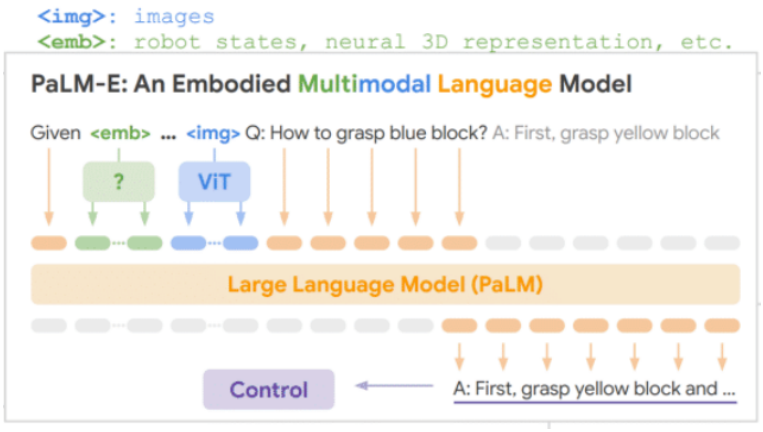
구체적으로 보면 perceiver resampler로부터 나온 출력은 LLM에서 아래와 같이 사용됨. cross attention과 FFW에서는 해당 연산 결과와 $\tanh(\alpha)$*를 곱한 값과 기존의 입력을 더한 것을 출력으로 함. 이때* tanh *gate는 cross attention에 따라 추가되는 imformation의 양을 조정하기 위한 것으로, 0으로 초기화되고 학습됨.*



3.2.2. Visual Token As Input : PaLM-E, VILA

1. PaLM-E

PaLM-E(Pathways Language Model with Embodied)는 google research에서 개발한 multi-modal model로, visual 등 임의의 modality를 단순 token화하고 LLM에 넣어 처리함.

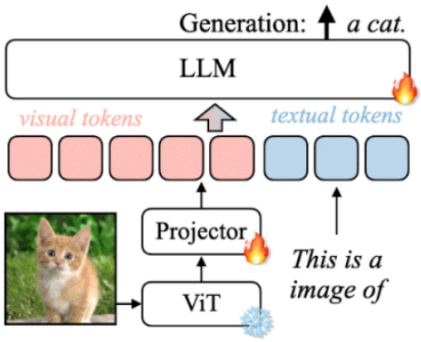


2. VILA

VILA(Visual Language Model, VLM)는 PaLM-E와 같이 단순 token화를 수행하는 model로, visual-language 처리에 특화되어 있음.

VILA는 아래와 같은 구조로 되어 있음. projector는 ViT(image)의 결과를 language에 사용하기 위해 변환을 수행하는 부분임. VILA는 image-language에 대한 alignment를 위해 우선 projector를 학습시

키고, 이후 pre-training을 수행한 뒤 SFT를 적용하는 식으로 학습시킴.



VILA와 관련된 특징들로는 아래와 같은 것들이 있음.

1) LLM으로 활용할 때 pre-training에서 model을 freeze했더니 괜찮은 zero-shot 성능을 보임. 하지만 이 경우 in-context 학습 능력이 떨어진다고 함.

2) pre-training 시에 단순히 image-text 쌍을 사용하는 것보다, image와 text를 interleave(번갈아)하게 사용하는 것이 더 좋은 성능을 낸다고 함.

3) fine-tuning 시에 image-text 데이터만을 사용하는 것보다, text-only 데이터를 함께 사용하면 text에 대한 처리 능력과 image-text 처리 능력이 모두 좋아진다고 함.

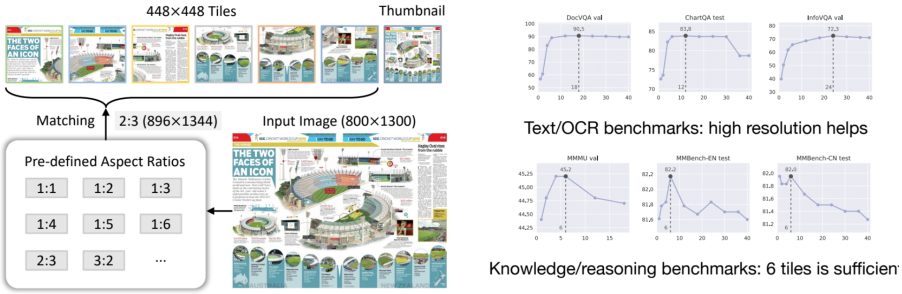
4) 원본 데이터의 resolution이 크면 model 성능이 좋아짐. 특이한 점은 token 개수보다 원본 데이터의 resolution이 성능이 영향이 더 큼. 예를 들어, 높은 resolution의 데이터에 downsample을 적용하여, token의 개수를 낮은 resolution의 token의 개수보다 적게 만들어도, 성능은 높은 resolution을 가지는 데이터가 더 좋다고 함. 물론 downsample하지 않은 데이터를 활용한 것의 성능이 가장 좋음.

3. High Resolution Image 처리

앞서 설명한 것처럼 원본 데이터의 resolution이 커지면 vision language model의 성능이 좋아질 수 있음. 하지만 resolution이 큰 데이터를 그대로 활용하면 연산량이 너무 많아지고, 그렇다고 downsample을 적용하면 성능이 떨어짐. 이에 따라 vision language model의 ViT에서 high resolution image를 처리하는 기법에는 아래와 같은 것들이 있음.

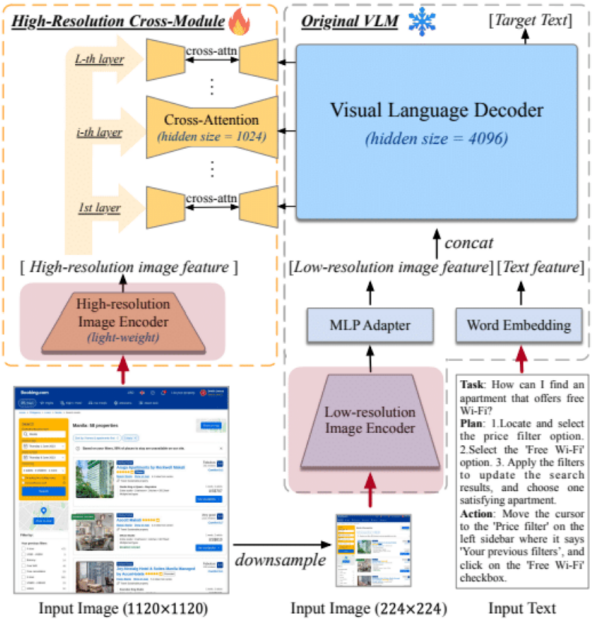
1) Tiling&Thumbnail(InternVL)

미리 정의된 template(ratio)에 맞춰 원본 high resolution image를 tiling하고, 원본 image의 작은 버전인 thumbnail을 하나 만들어 활용함. 이때 task에 따라 너무 만은 tile로 쪼개면 성능이 떨어지기도 함.



2) 2개의 ViT 활용

아래 그림과 같이 원본 image를 downsample(low resolution으로 변환)하고 token화하여 text와 함께 visual language decoder에 넣고, 원본 image는 별도의 incoder에 넣음. 이후 low resolution image와 text는 Q로 하고, 원본 image는 K와 V로 하는 cross attention 연산을 수행하여 출력을 도출함.



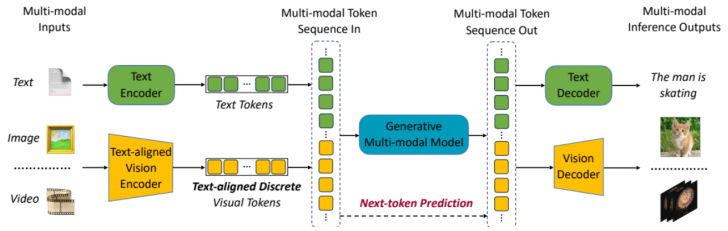
Flamingo에서는 image와 language가 asymmetric(비대칭)하게 처리되는데, PaLM-E에서는 이를 symmetric(대칭)하게 처리함.

3.2.3. VILA-U

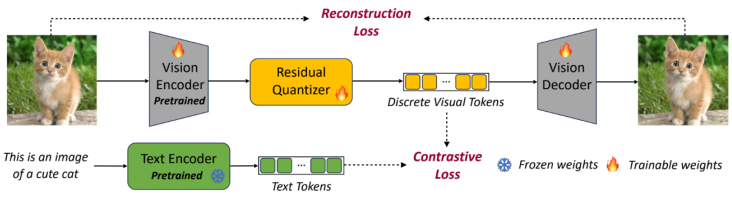
VILA-U(Unified Visual Language Model)는 video, image, language를 통합하여 understanding과 generation을 수행하는 단일 autoregressive model임.

단일 model이지만 거의 SOTA(State-of-the-Art, 최첨단)만큼의 성능을 낸다고 함.

VILA-U는 아래와 같은 구조를 가짐. 즉, encoder로 token화하고 generative model에 넣은 뒤, 그 결과를 사용하여 decoder로 출력을 생성함.



이때 구체적으로 visual 데이터의 처리에는 vision tower가 사용됨. Vision Tower는 image 입력을 discrete(quantization 적용) token으로 변환한 뒤 이를 text 입력과 align하는 부분으로, 아래와 같은 구조를 가짐. 학습 시에는 text와 align이 잘 되었는지를 나타내는 constrastive loss와, generation을 잘 수행하는지를 나타내는 reconstruction loss를 계산함.



3.3. Prompt Engineering

3.3.1. Prompt Engineering

Prompt Engineering은 model이 원하는 결과를 생성하도록 prompt를 잘 설계하고 입력하는 기법임.

단순하게는 입력을 구체적으로 적는 방법을 사용할 수 있고, 또한 아래와 같은 기법들을 활용할 수 있음.

1. Zero-shot Prompting
Zero-shot Prompting은 각 task 별로 하나의 model을 구성해 사용하던 방식과 달리, 하나의 foundation model로 추가적인 학습 없이 여러가지 task를 처리할 수 있도록 한 기법임.

2. Few-shot Prompting
Few-shot Prompting은 추가적인 학습 없이, 몇 가지 예시를 제공하여 어떤 새로운 task를 처리할 수 있도록 한 기법임. 이는 입력된 예시들의 맥락(context)를 활용하므로 In-context Learning이라고도 함.

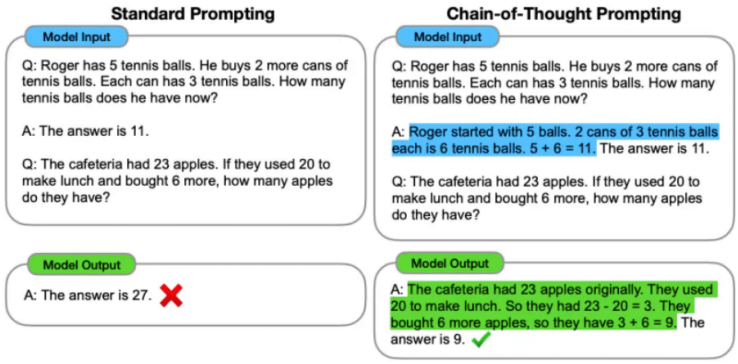
few-shot prompting 시에는 아래와 같은 경험적 노하우가 있다고 함.

1) classification에 대한 예시를 제공할 때는, 각 class에 대한 예시의 개수가 서로 같은 것이 좋다고 함.

2) text 예시 등을 제공할 때는, 각 예시에서의 format을 서로 맞추는 것이 좋다고 함.

3. Chain-of-Thought
Chain-of-Thought는 model이 여러 중간 단계를 거쳐 복잡한 추론 과정을 해결할 수 있도록 유도하는 기법임. 즉, thinking process를 알려주는 것임.

단순히 입력으로 "Let's think step by step."이라는 문장을 추가하기만 해도 복잡한 문제가 zero-shot으로 해결되기도 한다고 함.

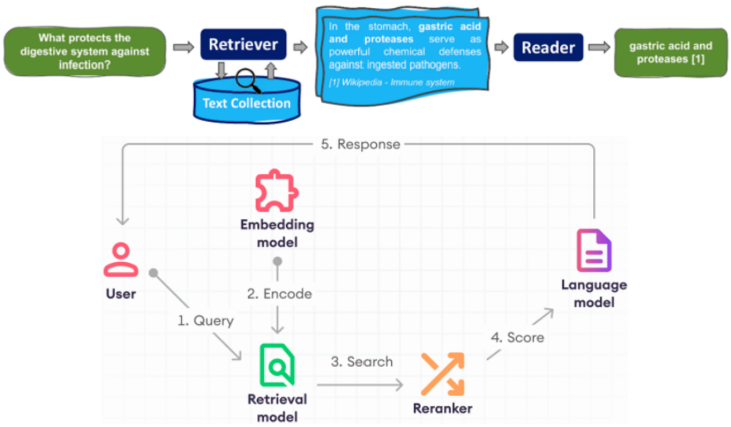


3.3.2. Retrival Augmented Generation

Retrival Augmented Generation(RAG)는 LLM 내부에 저장된 데이터만이 아니라 외부(DB 등)에 저장된 데이터를 활용하는 기법임.

LLM에 직접 데이터를 저장한다면, 해당 데이터는 유지 또는 수정되기 어려움.

아래와 같이 사용자로부터 query가 들어오면 embedding model을 활용해 데이터를 vector화하고, retriever로 query에 맞는 데이터를 선택한 뒤 LLM에 전달함. 또한 추가로 reranker를 사용해 query와의 관련성을 조정할 수 있음.



embedding model은 MTEB라는 benchmark를 사용해 평가된다고 함.

4. Long Context LLM

long context LLM에 대한 학습과 inference 시에는 더 적은 memory를 사용하면서 더 빠르도록 최적화하는 것이 중요함. 이와 관련하여 context length는 LLM이 한 번에 처리할 수 있는 입력의 길이를 의미함. 즉, 출력 생성 시에 고려할 수 있는 context의 크기임.

4.1. Context Extension

long context를 처리할 수 있도록 context length를 늘리는 방법을 알아보자.

4.1.1. RoPE

앞에서 다룬 RoPE를 적용하고 θ를 조정하여 더 긴 context를 처리하도록 할 수 있음. θ를 줄여(base를 늘려) context length(window)를 늘릴 수 있고, LLM은 fine-tuning하기 전에 context length를 우선 늘려야 한다고 함.

4.1.2. LongLoRA

LongLoRA는 long context LLM을 위한 LoRA 기반의 fine-tuning 기법임.

어떤 LLM이 처리할 수 있는 길이보다 긴 context를 처리하도록 하려면 fine-tuning을 적용해야 하는데, long context LLM을 위한 fine-tuning은 그 비용이 크므로 efficiency를 더 높일 필요성이 있음. 이때 LoRA를 단순 적용하면 성능 저하가 심하고, LongLoRA를 적용하면 성능을 유지할 수 있음.

LongLoRA에서는 1) shift sparse attetion과 2) enhanced LoRA를 활용해 efficiency를 확보함.

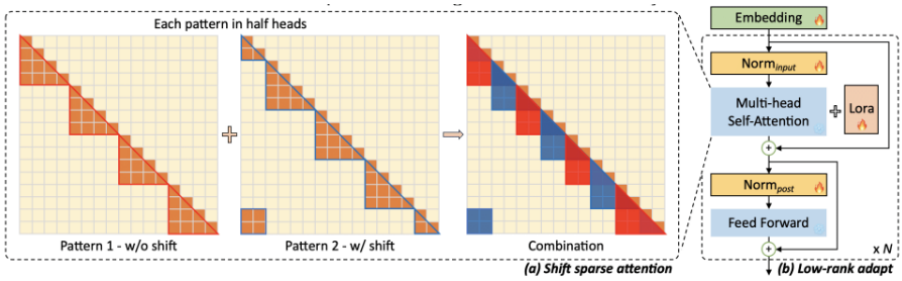
1. Shift Sparse Attention
Shift Sparse Attention은 token을 여러 group으로 묶어 attention 연산을 적용(sparse attention)한 결과와, token을 shift한 뒤 group으로 묶어 attention 연산을 적용한 결과를 모두 활용하는 기법임.

attention 연산은 각 token마다 나머지 token들에 대한 연산을 수행하므로 입력(token 개수)에 따른 복잡도가 n^2이고, long context LLM에서의 병목이 됨.

아래 그림과 같이 아래쪽 삼각형 전체를 연산해야 했던 기존의 attention 연산과는 달리, 해당되는 group 내부의 token들끼리만 attention 연산을 수행함. 기존 방식에 비해 sparse하므로 이를 Sparse Attention이라고 함.

하지만 완전히 각 group을 따로 연산하면 group끼리는 context가 반영되지 않으므로, token을 shift하여 sparse attention을 적용한 결과를 함께 활용함. 즉, multi-head중 절반의 head에 대해서는 단순 sparse attention을, 나머지 절반에 대해서는 shift한 뒤 sparse attention을 적용함. 이에 따라 서로 다른 group

에 대해서 context가 반영된 결과가 도출되고, 이런 shift sparse attention layer를 여러 개 쌓으면 바로 옆 group만이 아니라 전체로 context가 반영되게 됨.



또한 학습 시에는 sparse attention을, inference에는 full attention을 사용하는 것이 성능이 더 좋다고 함.

2. Enhanced LoRA
LongLoRA에서는 input embedding layer와 normalization layer도 fine-tuning함. 즉, embedding/normalization layer의 parameter들을 학습 가능하도록 지정한 뒤 fine-tuning함. embedding/normalization layer에 대해서는 parameter 개수가 적으므로 싸게 tuning할 수 있음.

sparse attention뿐만 아니라 embedding/normalization layer에 대한 fine-tuning까지 적용해야 LoRA에 해당하는 부분에서 낮은 rank를 쓰면서 성능이 full fine-tuning만큼 나옴.

shift sparse attetion layer에 대해서 layer별로 cross한 것도 있지만 head별로 cross한 것이 성능이 더 좋다고 함.

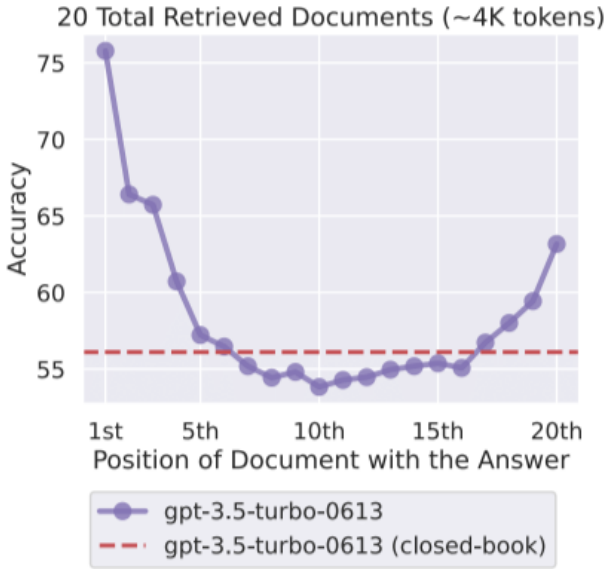
positional encoding을 함께 고려하여 context length를 더 확장할 수 있다고 함.

4.2. Evaluation of Long Context LLM

long context LLM을 평가하는 방법을 알아보자.

4.2.1. Lost-in-the-Middle Phenomenon

Lost-in-the-Middle Phenomenon은 long context에 대해 관련 정보가 context의 양 끝에 있는 경우에 비해 중간에 있는 경우에 accuracy가 떨어지는 현상임. 즉, 성능 그래프를 보면 아래와 같이 u자임.



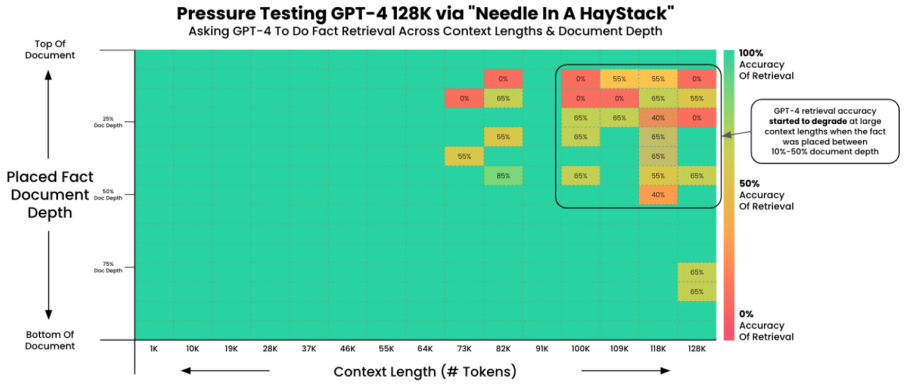
즉, LLM이 long context에 대해 괜찮은 답변을 내놓고 있다고 해도 실제로 모든 부분에 대한 처리가 잘 이뤄지지 않았을 수 있음. 이 경우 perplexsity만을 지표로 활용하면 중간에서 성능이 낮아도 양쪽 끝 부분에 의해 성능이 좋게 나올 수 있음. long context LLM을 평가할 때는 이를 고려한 엄격한 평가가 필요함.

4.2.2. Long Context Benchmarks : NIAH, LongBench

long context를 처리하는 model을 평가하는 benchmark로는 NIAH와 LongBench가 있음.

1. NIAH
NIAH(Needle In A Haystack)는 긴 문서(haystack)의 중간에 짧은 정보(needle)을 삽입해 model에 입력하고, 해당 needle을 찾도록 하는 benchmark임.

아래와 같이 여러 depth(needle을 넣는 위치)로 model을 평가함. 즉, Lost-in-the-Middle Phenomenon을 고려한 평가 방식임.



2. LongBench
LongBench는 6가지 task에 대한 21개의 데이터셋을 포함하고, 최대 13000개의 token을 가지는 long context를 입력으로 활용해 F1, ROUGE 등으로 평가하는 benchmark임.

LongBench에 의하면 long context에 특화되지 않은 model에 적절한 PE 등을 적용하는 것이 long context 처리 능력을 향상시켜 주지만, 애초에 long context를 위해 설계된 model보다는 long context에 대한 성능이 떨어진다고 함.

4.3. Efficient Attention Mechanism

long context에 대한 inference를 개선하는 방법을 알아보자.

4.3.1. StreamingLLM

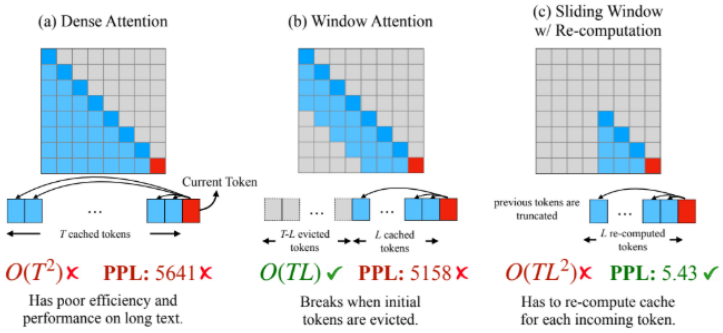
StreamingLLM은 attention sink를 고려하여 적은 computation과 memory를 사용하면서 long context에 대한 inference를 개선함.

1. 기존 방식의 한계
긴 상호작용이 요구되는 streaming(실시간 동작) application에서는 처리해야 하는 context length가 커질수록 decoding stage에서 memory 사용량이 너무 커지게 됨.

아래와 같이 단순 attention은 long context에 대해 너무 많은 computation과 memory가 필요함.

Window Attention은 window를 사용하여, 최근 token들에 대해서만 KV cache에 값을 저장해 두고 attentino 연산에 활용하는 방식임. 당연하게도 window를 벗어난 부분은 참조하지 못하는데, initial token이 KV cache에서 나가는 순간 성능이 급격히 떨어짐.

Re-computation을 활용한 Sliding Window는 window attention과 유사하지만, window 내의 token들에 대해 매번 attention 연산을 새로 수행함. perplexity는 비교적 낮지만, computation이 너무 많음.



2. Attention Sink

Attention Sink는 initial(첫 번째) token이 실제 의미적 중요도에 상관없이 높은 attention score를 가지는 현상임.

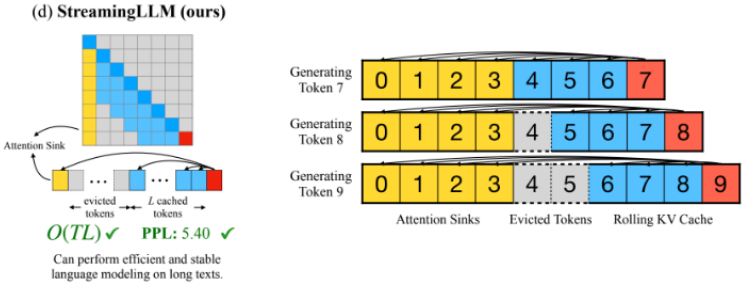
이는 attention 연산에서는 softmax에 의해 대상의 개수가 적더라도 총합이 1이고, inital token은 다른 모든 token에 대한 연산에서 활용되기 때문에 발생하는 현상임. 실제로 initial token을 아무 의미가 없는 token(공백 문자)으로 변환해도 높은 attention score를 가진다고 함. 즉, initial token은 그 위치에 의해 다른 token들에 대해 attention score가 높게 계산되고, 이에 따라 initial token이 attention 연산에서 제외되는(KV cache에서 제거되는) 경우 model의 성능이 급격히 떨어짐.

3. StreamingLLM의 아이디어

이에 따라 StreamingLLM에서는 맨 처음 4개의 token들(Sink Token)을 KV cache에서 제거하지 않고 attention 연산에 계속 활용하는 것으로 long context에 대해 computation, memory를 적게 사용하면서 성능(perplexity)을 유지함. 이때 sink token의 개수는 4개가 경험적으로 제일 좋다고 함. sink token의 활용은 paged attention 등을 적용하여 KV cache의 physical memory만 바꿔주는 식으로 구현할 수 있음.

4개의 sink token은 pre-trained model에 대해 선정한 것이었는데, 대신 하나의 learnable sink token을 pre-training 전에 추가하여 학습시키는 것으로 설계할 수도 있다고 함.

또한 decoder model뿐만 아니라 encoder model에도 attention sink가 존재한다고 함.



하지만 이 경우에도 window가 이동하며 제외(evict)된 token들은 attend될 수 없으므로, context가 길어지면 sink token을 제외하고는 evicted token들에 대한 정보는 사라짐. 즉, window 크기의 한계에 의해 성능이 급격히 떨어지게 되는데, DuoAttention으로 이를 해결할 수 있음.

4.3.2. DuoAttention

DuoAttention은 두 가지 종류의 attention을 활용하여 memory/latency 측면에서의 병목을 해결하고 정보 손실을 줄이면서 long context를 처리하는 기법임.

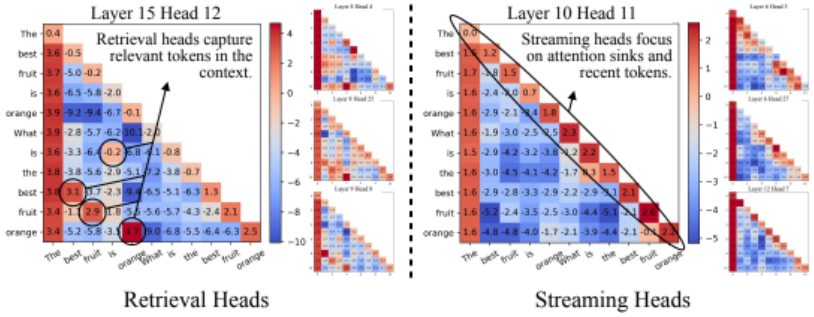
long context에 대한 처리는 각 token에 대한 KV값을 저장해야 하므로 memory 측면에서의 병목도 존재하고, 각 token에 대한 정보를 참조하여 decoding을 수행하므로 context length에 비례해 latency가 증가한다는 병목도 존재함. 또한 KV cache에 대한 기존의 compression 기법들은 LLM의 long context 처리 능력을 떨어뜨림. DuoAttention은 이런 memory와 latency 측면에서의 병목을 개선함.

1. Retrivial Head vs. Streaming Head

DuoAttention에서는 transformer의 각 head가 retrivial head와 streaming head로 구분됨. 즉, 두 가지 종류의 attention 연산을 수행함.

Retrivial Head는 모든 이전 token에 대해 attention(full attention)을 수행하는 head임. 즉, 이전 token들 중 문맥상 중요한 token을 활용하게 됨. retrivial head에서의 KV cache를 compression하면 성능 저하가 발생함.

Streaming Head는 attention sink token과 최근 token들에 대해서(window 사용)만 attention을 수행하는 head임. streaming head에서의 KV cache는 compression해도 성능 저하가 심하지 않음.



2. 핵심 아이디어

DuoAttention에서는 모든 head에서 full attention을 수행할 필요가 없다고 가정함. 이에 따라 일부 head는 문맥적으로 중요한 token을 반영하며 큰 KV cache를 활용하는 retrivial head로, 나머지는 sink token과 최근 token을 반영하며 작은 고정된 KV cache를 활용하는 streaming head로 사용함.

이때 어떤 head를 retrivial/streaming head로 사용할지는 trainable gate value인 α를 사용한 최적화 기반 기법으로 결정함. 즉, 우선 어떤 head(i번째 layer, j번째 head)에 대해 아래와 같이 attention을 정의함. 이때 $M_{casual}, M_{streaming}$은 retrivial/streaming 각각에 대한 mask임.

$$full\_atten = softmax(QK^T \cdot M_{casual})V$$

$$streaming\_atten = softmax(QK^T \cdot M_{streaming})V$$

$$attn_{i,j} = \alpha_{i,j} \cdot full\_atten + (1 - \alpha_{i,j}) \cdot streaming\_atten$$

이후 아래와 같이 loss function을 정의하여 α를 최적화함. 즉, full attention을 적용했을 때의 출력과, 위와 같이 α를 활용해 두 가지 attention을 섞었을 때의 출력 사이의 차이를 활용함(distillation). 또한 full attention의 사용이 최소화되는 것이 좋으므로 α의 절댓값 또한 활용함.

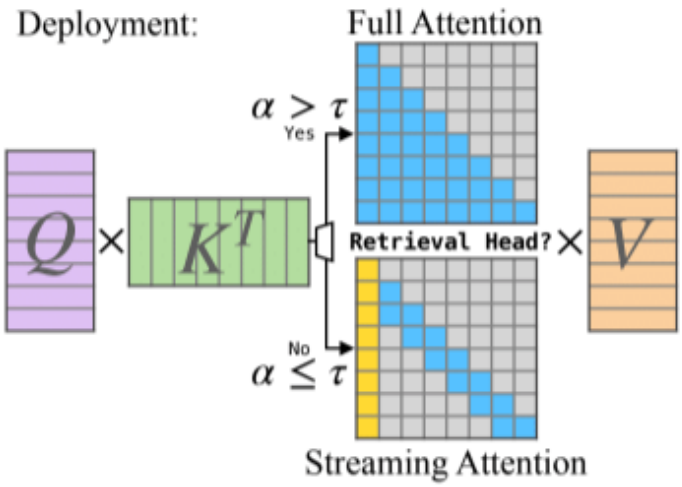
α에 대한 최적화 과정에는 NIAH와 유사하게 긴 context 중간에 passkey를 삽입한 synthetic(합성) 데이터셋을 활용함. model에 따라 다르겠지만 Llama2-7B의 경우 32개의 layer에서 각각 32개의 head를 사용하므로 1000개 가량의 α만 학습시키면 되고, 이는 A100 gpu를 활용했을 때 몇 시간이면 완료된다고 함.

$$\mathcal{L}_{\text{distill}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=T-l+1}^{T} (\boldsymbol{H}_{\text{full}}^{(i)}[j] - \boldsymbol{H}_{\text{mixed}}^{(i)}[j])^2$$

$$\mathcal{L}_{\text{reg}} = \sum_{i=1}^{L} \sum_{j=1}^{H} |\alpha_{i,j}|$$

$$\mathcal{L} = \mathcal{L}_{\text{distill}} + \lambda \mathcal{L}_{\text{reg}}$$

이후 inference time에는 threshold τ를 각 head의 α와 비교해 해당 값보다 크면 retrivial head로, 작으면 streaming head로 활용함. 당연하게도 실제 연산 이후 각 head의 출력은 종합됨.



또한 DuoAttention의 inference는 KV cache에 값을 채워넣는 등의 작업을 수행하는 pre-filling stage와 최종 출력을 생성하는 decoding stage로 나뉨. 이때 pre-filling stage에서는 retrivial/streaming head 모두에 대해 chunking(FlashAttention-2)을 적용한다고 함.

이후 4bit KV cache quantization을 함께 적용하여 더 개선할 수 있다고 함.

4.3.3. Quest

Quest는 query-aware attention을 활용하여 inference에서의 연산량을 줄이면서 성능을 유지함.

accuracy, latency, efficiency 측면에서 full cache(full attention)보다 뛰어남.

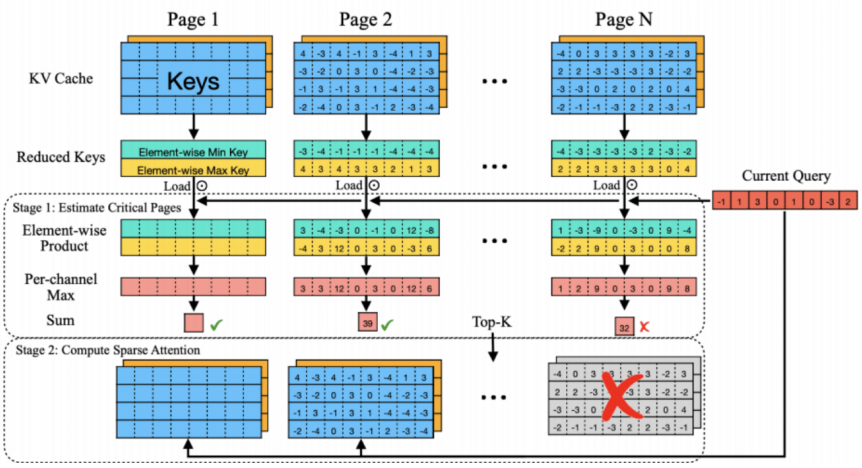
1. 기존 방식의 한계
full attention은 모든 token에 대해 attention을 수행하므로 연산이 오래 걸리고, StreamingLLM과 같이 고정된 방식으로 token을 evict하는 기법은 연산 시간은 줄이지만 evict된 token을 다시 참조할 수 없어 성능 저하가 발생함.

특히 특정 시점에 중요도가 낮아 evict된 token도 이후에 중요도 높게 활용될 수 있음. 즉, 현재 처리하는 query에 따라 이전 token의 중요도가 가변적임. 이에 따라 Quest에서는 query-aware attention으로 이전 token들 중 중요도가 높은 것을 활용하여 복잡도를 낮추면서 성능을 유지함.

2. 핵심 아이디어
Quest에서는 KV cache 전체를 GPU memory에 저장하고, query와의 연관성(중요도)이 높은 부분만을 attention에 활용하여 memory로부터의 data 이동을 줄여 inference를 가속함.

구체적으로는 아래와 같이 KV cache를 여러 개의 page로 나누고, 현재 query와 비교하여 중요도가 높은 page만 fetch하여 attention 연산에 활용함.

이때 page의 모든 부분과 연산을 수행하는 것은 overhead가 크므로 아래와 같이 계산함.
1) K값을 element-wise하게 최솟값과 최댓값을 계산해 각각에 대한 벡터를 얻음.
2) 두 벡터에 대해 query와 element-wise하게 곱셈을 수행함.
3) 두 벡터의 값에 대해 per-channel로 최댓값을 계산하여 하나의 벡터를 도출함. 이 값은 attention 연산에 대한 상한을 나타냄.
4) 해당 벡터의 값 전부에 대한 합(score)을 계산함. 이 값이 해당 page의 중요도를 나타냄.
5) 합의 값이 큰 page들만 fetch함.



이에 따라 KV cache에 새로운 token이 들어올 때 최솟값과 최댓값을 계산해둠. 이후 inference 시에는 해당 최솟값과 최댓값을 활용해 score를 계산함.

정리한 것처럼 Quest는 KV cache 자체를 줄이는 것이 아니라 query-aware attention을 통해 inference를 가속화하는 기법임.

4.4. Transformer 이후의 기법들

transformer가 아니면서도 efficient한 아키텍처들을 살펴보자.

4.4.1. SSM

1. Mamba

Mamba는 SSM(State-Sapce Model)을 활용한 긴 sequence를 선형 시간복잡도로 처리함.

transformer에서는 token들 사이의 context를 반영하는 부분(attention)과, 개별 token에 대한 연산을 수행하는 부분(feed forward)으로 나뉨. Mamba에서는 attention 연산을 SSM을 활용하여 대체함.

SSM은 아래와 같이 동작함. Mamba에서는 이를 수정하여 Selective SSM으로 활용함. selective SSM은 각 state가 해당 token(입력)의 요구에 맞춰 지정되는 방식임. 즉, 기존의 SSM에서는 아래의 행렬 A, B, C가 x에 대해 독립적으로 결정되었는데, selective SSM에서는 x에 의해 결정됨.

State (h): Represents the current knowledge about the sequence.
Input (x): New information entering the sequence.
Update: The state adjusts dynamically based on input, focusing only on relevant data.

Linear Recurrence

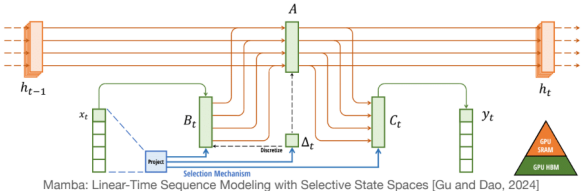
$$h_t = \overline{A}h_{t-1} + \overline{B}x_t$$
$$y_t = Ch_t$$

$=$

Global Convolution

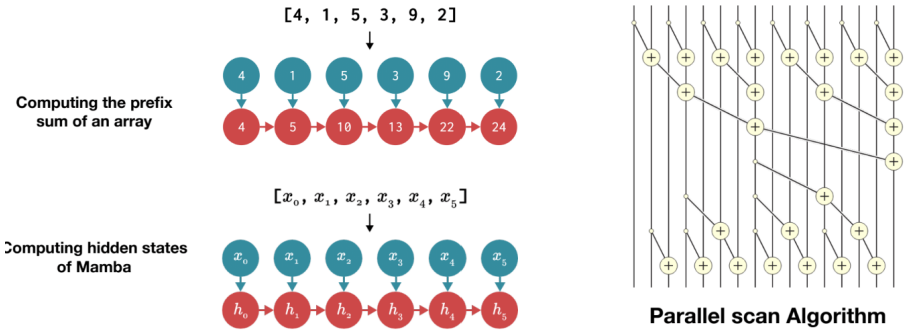
$$\overline{K} = (C\overline{B}, C\overline{AB}, \ldots, C\overline{A}^k\overline{B}, \ldots)$$
$$y = x * \overline{K}$$

A controls state transition: *How should I forget or update the state over time?*
B maps new input to output: *What part of the new input should I remember?*
C maps state to output: *How can I use the state for a good prediction?*



Mamba: Linear-Time Sequence Modeling with Selective State Spaces [Gu and Dao, 2024]

기존의 SSM이 가지는 이점은 해당 연산이 x에 대한 convolution 연산으로 취급되어 각 행렬을 precompute할 수 있다는 것이었는데, selective SSM에서는 행렬이 x에 종속되므로 precompute가 불가능함. 대신 parallel scan 알고리즘을 사용하면 병렬처리 가능하다고 함.



2. Jamba

Jamba는 transformer와 Mamba, MoE(Mixture-of-Experts)를 결합하여 성능과 efficiency 사이의 균형을 맞춘 하이브리드 방식임.

5. Vision Transformer

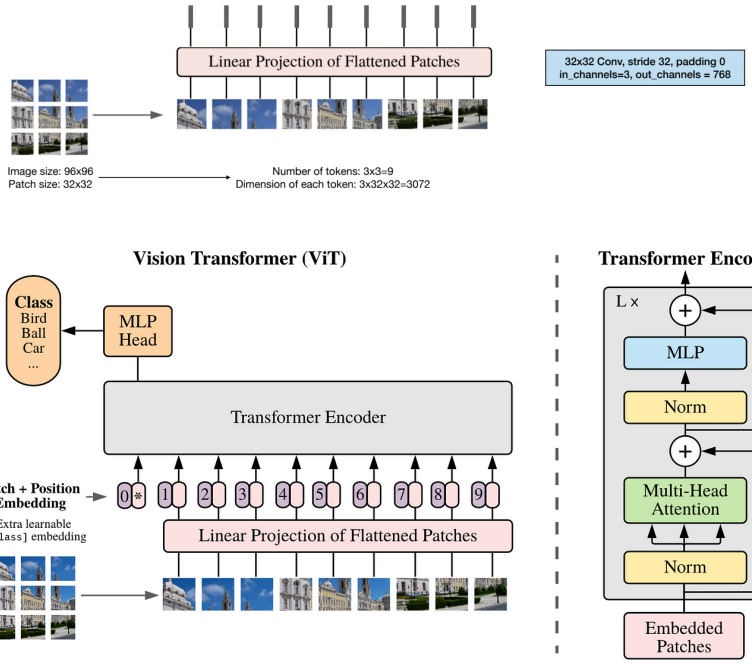
5.1. ViT

5.1.1. ViT

ViT(Vision Transformer)는 vision 처리를 위한 transformer임.

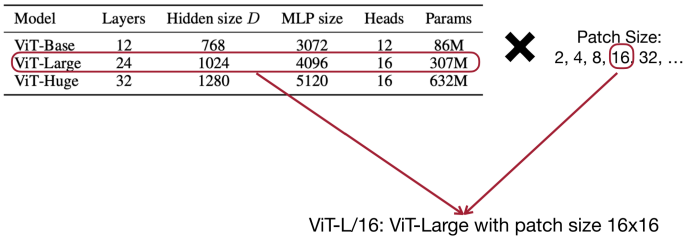
transformer에서는 token 단위로 데이터를 처리함. 이때 language에 대해서는 단어 또는 그에 준하는 단위를 쓰면 되므로 token으로의 변환이 단순함. image에 대해서도 적절한 방법을 적용해 tokenize하면 transformer에 입력으로 넣을 수 있음.

vision data를 처리하는 자연스러운 방법은 image를 여러 patch로 쪼개서 token화하는 것임. 즉, image를 patch의 sequence로 처리할 수 있음. 이후 아래와 같이 각 patch를 flattern한 뒤 사용하는 transformer의 차원으로 linear projection하면 되는데, 이때 FC를 쓸 수도 있지만 아래와 같이 convolution layer를 사용하는 것이 일반적인 방법임. 이후 생성된 token을 transformer(encoder)에 넣어 처리할 수 있음.



ViT에는 아래와 같이 model의 구성과 patch size에 따라 여러 variant가 존재함. 당연하게도 patch size가 클수록 더 coarse(더 큰 단위로 처리)한 것임.

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M



ViT는 학습 데이터가 적으면 CNN보다 성능이 안 좋지만, 학습 데이터가 많아지면 CNN보다 성능이 좋아짐.

5.2. Efficieny/Acceleration on ViT

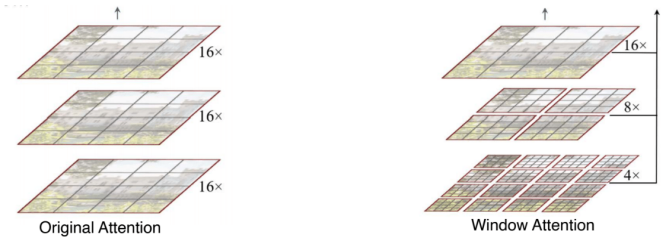
ViT를 어떻게 더 efficient하도록 수정할 수 있을까? 특히 image 처리에는 높은 resolution의 fine-grained 처리가 요구되는 경우가 많고(의학, 애니메이션, 실시간 인식 등), 또한 resolution이 높을수록 accuracy 또한 높음. 하지만 그에 따라 더 많은 computation이 필요하므로 efficiency의 중요성이 큼.

5.2.1. Window Attention

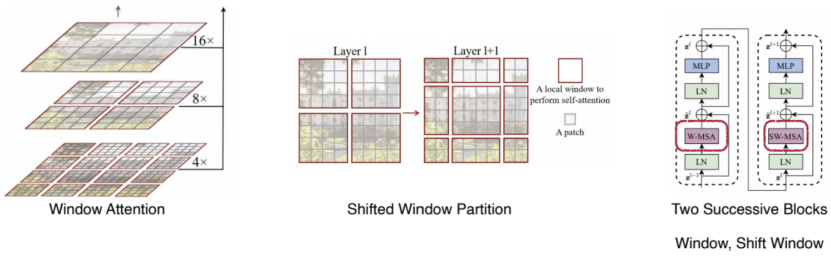
1. Window Attention

Window Attention은 모든 patch가 서로를 참조하여 attention 연산을 수행하는 기존의 방식과 달리, 정해진 window 내부의 token들에 대해서만 연산을 수행하는 방식임.

구체적으로는 아래와 같이 각 attention에 대해 window 크기를 점점 크게 하는 등으로 지정하여 적용함. 여러 기존의 ViT에서는 attention 연산에 의해 n^2의 시간복잡도를 가졌지만, window에 들어가는 token의 개수가 고정되어 있으므로 n(linear)의 시간복잡도를 가지게 됨.



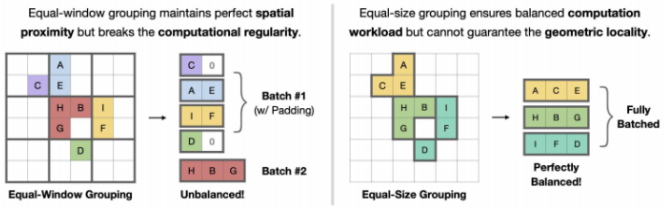
Swin Transformer에서는 아래와 같이 ViT에 window attention을 적용함. 이때 단순 window attention 뿐만 아니라, window를 shift하는 Shifted Window Attention 또한 사용함. 이는 단순 window attention에서는 token들이 자신이 포함된 window 내부의 token들끼리만 정보공유가 가능한 문제가 발생하기 때문임.



2. Sparse Window Attention

처리하는 image의 특징에 따라서 window attention에서 window가 중요한 것은 아니므로 sparsity를 활용할 수 있음.

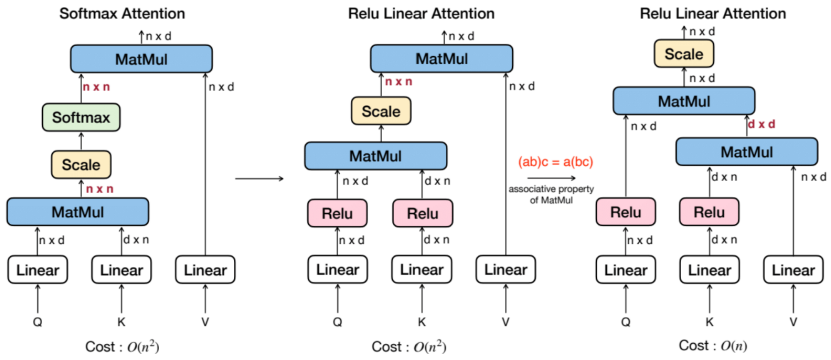
아래와 같이 각 window의 크기를 균일하게 설정하고 해당 window 내부에 존재하는 데이터를 활용할 경우, window마다 포함하는 데이터의 수가 달라서 효율적인 연산이 어려움. 이에 따라 데이터의 수에 따라 grouping하여 연산할 수 있음.



5.2.2. Linear Attention

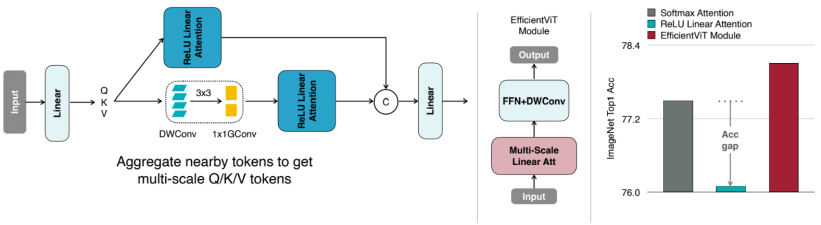
Linear Attention은 softmax 대신 ReLU를 사용하고, 연산 순서를 조정하여 n^2의 복잡도를 가지는 기존의 attention 연산을 n의(linear) 복잡도를 가지도록 개선한 기법임.

아래와 같이 구성함. softmax를 ReLU로 바꾸기만 하면 복잡도가 그대로지만, softmax를 대체했으므로 연산 순서를 뒤바꿔 computation을 줄일 수 있음. 이때 n은 sequence length이고, d는 embedding 차원임. 이를 고려해 복잡도를 계산해 보면 softmax attention과 ReLU linear attention은 각각 행렬 곱에 의한 곱셈 연산이 dn^2, d^2n만큼씩 발생하므로 각각 복잡도가 n^2, n임.



하지만 linear attention은 computation이 줄어든 만큼 성능 저하가 발생함. softmax attention에 비해 linear attention은 중요한 정보에 대한 attention score가 높게 계산되지 못하고, 이에 따라 score의 분포가 sharp하지 못함(강의에서는 이를 weak하다고 표현함.). 그래서 linear attention은 전반적인 context를 파악하는 건 잘 해도 특정 부분에 대한 정보는 잘 잡아내지 못함.

이에 따라 아래와 같이 convolution 연산은 특정 부분에 대한 정보를 뽑아내는 데에 효과적이므로 이를 branch로 활용할 수 있음. 이때 efficiency 유지를 위해 depth convolution을 활용함. 이렇게 하면 성능도 좋아지고, 더 빨라진다고 함.



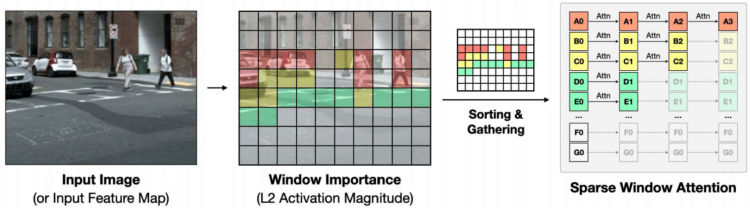
5.2.3. Sparse Attention : SparseViT

SparseViT(Sparse Vision Transformer)는 window별 중요도를 계산하는 등의 방식으로 sparsity를 확보하여 image 처리를 하는 기법임.

resoluton에 따른 pixel 수는 n^2으로 증가하는데, 증가된 resolution만큼 sparsity를 활용하면 동일한

크기의 activation으로 high resolution image를 처리할 수 있음.

SparseViT에서는 입력 activation을 여러 window로 쪼갠 뒤, 각 window에 대해 L2 norm으로 크기(magnitude)를 계산하여 그 값이 클수록 중요도가 높은 것으로 평가함. 이후 아래와 같이 중요도를 기준으로 정렬한 뒤 여러 layer를 거치며 점진적으로 더 prune함.



이때 pre-trained model은 dense한 activation으로 훈련되는데, 각 iteration마다 layer별 sparsity를 무작위하게 결정하여 적용한 것으로 fine-tuning한다고 함. 또한 evolutionary search를 활용해 layer별 최적의 sparsity를 찾는다고 함. 자세한 내용이 궁금하면 논문을 읽어보자.

SparseViT를 적용한 후 실제로 image를 확인해 보면 관련성이 적은 배경 등이 prune된다고 함.

이런 아이디어는 language model에 대해서도 적용될 수 있다고 함.

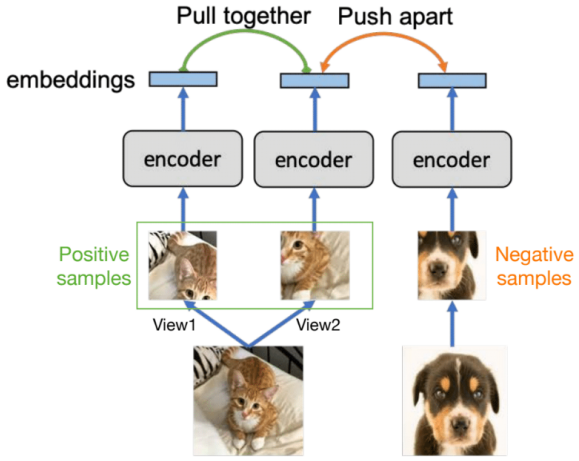
5.3. Self-supervised Learning for ViT

ViT는 데이터가 많을 때 성능이 좋아지지만, 데이터를 labeling하는 것은 비용이 많이 듦. 이에 따라 데이터에 대한 labeling 없이 학습을 수행하는 self-supervised 기법을 알아보자. 여기에서 Self-supervised(자기지도) learning은 데이터에 대한 별도의 label 없이, model이 자체적으로 학습 목표를 만들어 학습하는 방식임.

5.3.1. Contrastive Learning

1. Contrastive Learning
Contrastive Learning(대조학습)은 유사한 입력끼리는 거리가 가까워지도록, 상이한 입력끼리는 거리가 멀어지도록 loss를 계산하는 self-supervised 기법임.

구체적으로는 아래와 같이 positive sample과 negative sample로 구성된 데이터셋을 활용하여 각 embedding을 생성하고 비교함. 이때 두 embedding 사이의 거리는 유클리드 거리, 코사인 유사도 등을 활용하여 계산할 수 있음.

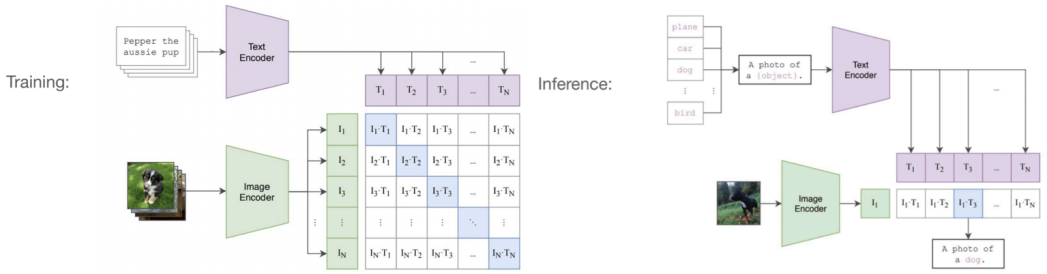


supervised ViT에 비해 self-supervised ViT의 성능이 더 좋다고 함.

2. Multimodal Contrastive Learning : CLIP
CLIP(Contrastive Language-Image Pre-training)은 각 입력에 대한 embedding을 생성한 뒤 코사인 유사도를 계산하여 일치하는 입력들에 대한 값이 최대가 되도록 학습시킨 language-image multimodal

model임.

즉, 아래와 같은 단순한 구조로 학습을 수행함. 물론 이때 학습을 위해 image-text 쌍이 데이터셋으로 존재해야 함. 이런 식의 학습 이후에는 zero-shot으로도 성능이 좋다고 함. 이후 inference에는 입력과의 코사인 유사도를 계산해 가장 높은 유사도를 가지는 것을 출력함.



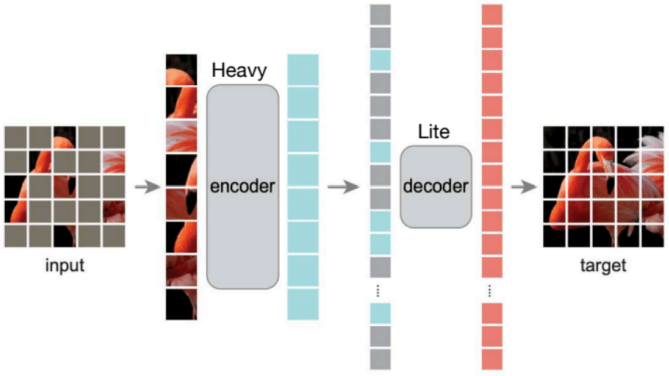
5.3.2. Masked Image Modeling : MAE

1. MIM

Masked Image Modeling(MIM)은 입력 image의 일부를 masking한 뒤 이를 복원하도록 하는 과정으로 학습시키는 self-supervised 기법임. MIM의 대표적인 예시로는 MAE가 있음.

2. MAE

MAE(Masked Autoencoders)는 아래와 같이 encoder와 decoder라는 부분(둘 다 transformer의 encoder model임.)을 활용하여 MIM으로 학습시키는 model임. 이때 encoder는 적은 부분으로 전체에 대한 정보를 추출해야 하므로 heavy하고, decoder는 해당 정보로 복원만을 수행하므로 lite함. 학습 이후에는 decoder 부분은 제거하고 encoder 부분만을 활용함.



실험 결과 원본 image에 대한 masking을 75%까지 적용해도 성능 저하가 없다고 함. 참고로 BERT에서의 최대 masking ratio가 15%인데, image는 text에 비해 redundancy가 많기 때문에 이 75%까지의 masking이 가능한 것으로 유추할 수 있음.

이는 BERT 등의 language model로부터 가져온 아이디어임.

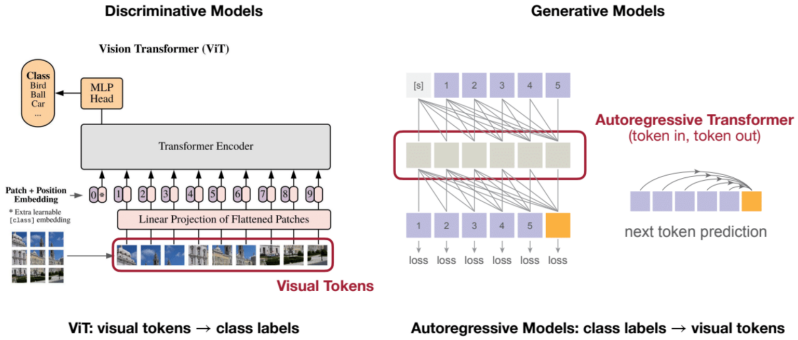
5.4. ViT & Autoregressive Image Generation

5.4.1. Autoregressive Image Generation

1. Autoregressive Image Generation

transformer를 활용하여 autoregressive하게 image generation을 수행하는 generative model을 구성할 수 있음. 여기에서는 이를 간단히 AR(autoregressive) model이라고 함.

transformer model은 아래와 같이 discriminative(판별) model과 generative model로 나눌 수 있음. 이때 generative model은 autoregressive하게 token을 생성함. 앞에서는 discriminative model인 ViT를 주로 다뤘지만, image를 생성하는 AR model도 구성할 수 있음. 즉, 이때의 AR model은 입력으로 language token을 받아서 출력으로 image token을 도출함. AR model은 실제로 multimodal에서 성능이 좋고, diffusion model보다 빠르게 구현할 수 있다고 함.



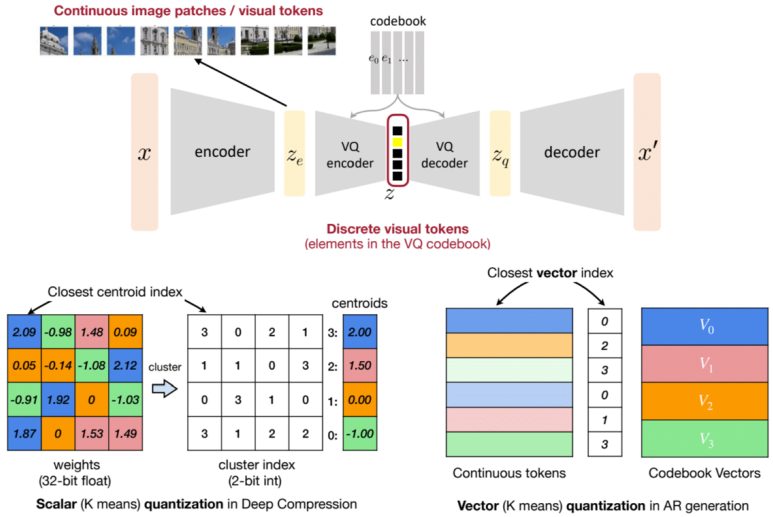
AR model에서 image token을 생성하는 방법은 아래와 같이 3가지로 분류할 수 있음. 즉, 순서대로 또는 임의의 모든 앞쪽 token을 참조하거나, 특정 부분의 token을 참조해 동시에 여러 token을 생성할 수 있음. 이때 모든 앞쪽 token을 모두 참조하는 것은 memory bounded인데, 특정 부분을 참조하는 것은 전체를 참조하는 방식보다 memory를 덜 사용함.

2. Vector Quantization

Vector Quantization(VQ)는 k-means-based quantization과 유사한데, 이를 벡터 단위로 적용하는 기법임. AR model에서는 image token에 대한 생성에 VQ를 사용함.

lanugage에서는 입력 데이터의 종류(vocabulary)가 이산적으로 고정되어 있지만, image에서는 각 pixel의 값이 연속적임. 이에 따라 language에 비해 image가 supervised learning이 어려움. vector quantization을 사용하면 image token을 discrete하게 변환하여 language와 image를 잘 섞고 generative model을 구성할 수 있음.

VQ에서는 k-means-based quantization과 유사한 방식으로 codebook을 사용하여 벡터 단위로 quantization함. 즉, 벡터에 대한 인덱스 데이터를 추출함. 이에 따라 image token를 생성하는 작업은 해당 인덱스를 생성하는 작업이 됨.

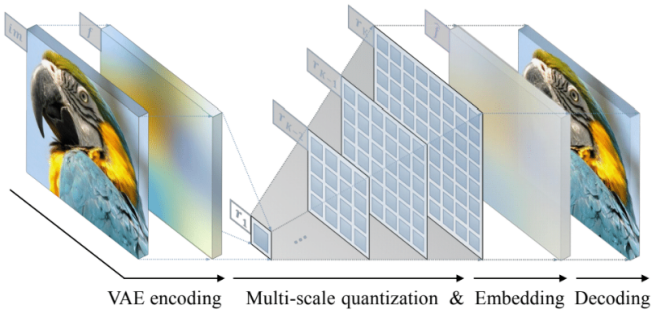


3. VAR

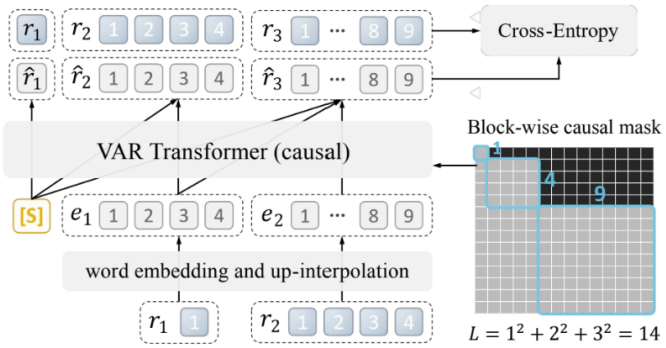
VAR(Visual Autoregressive Modeling)은 image 생성을 위한 AR model임.

VAR에서는 VQ를 활용하고, 아래와 같이 하나의 token으로부터 시작해 매 입출력마다 그 크기를 늘려가며 token을 생성함. 즉, 생성한 token들을 입력으로 넣으면 더 많은 개수의 token이 출력됨. 이에 따라 attention mask에 의한 attention map이 삼각형이 아니라 사각형들에 의한 모양이 됨. 동시에 여러 개의 token을 생성하므로 memory bounded에서 벗어남.

Stage 1: Training multi-scale VQVAE on images
(to provide the ground truth for training Stage 2)



Stage 2: Training VAR transformer on tokens
([S] means a start token with condition information)



사실 실제로 image는 연속적이므로 당연하게도 이런 discrete tokenizer는 한계가 있음. 생성한 image를 보면 퀄리티가 좋지 않음. HART는 이를 보완한 기법임.

5.4.2. HART

HART(Hybrid Autoregressive Transformer). diffusion model을 학습한 이후 정리한다.

6. GAN, Video, Point Cloud

domain specific 기법들에 대해서 살펴보자. 이런 domain들은 computation 비용이 크고, redundancy가 많으므로 efficiency 측면에서의 개선이 중요함.

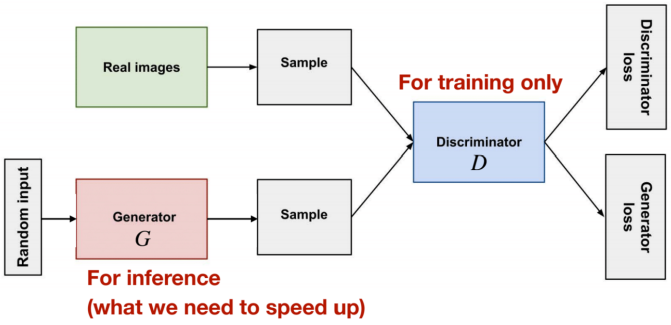
6.1. Efficient GAN

6.1.1. GAN

1. GAN

GAN(Generative Adversarial Network)은 데이터를 생성하는 generator와, 실제 데이터와 생성된 데이터를 구별하는 discriminator가 경쟁하며 학습하는 generative model임. 주로 image generation에서 많이 사용된다고 함.

구체적으로 GAN은 아래와 같은 구조를 가짐. 이때 discriminator는 학습 시에만 사용되고, inference 시에는 generator만 사용함. 이에 따라 GAN에 대한 compression은 generator에 적용하게 됨.



GAN은 효과적이지만, 이런 generative model은 model size, computation 등의 측면에서 비용이 큼. 이에 따라 compression을 잘 적용하는 것이 중요함.

2. Loss Function

GAN의 loss function은 아래와 같음. 여기에서 z는 random input(noise)이고, x는 real image임. 또한 θ는 generator의 parameter이고, ϕ는 discriminator의 parameter임. 실제 데이터는 1을 label로, 가짜 데이터는 0을 label로 함. 즉, generator는 $D(G(z))$가 1에 가까워지도록(real image로 판단되도록) 학습되고, discriminator는 $D(x)$가 1에 가까워지고 $D(G(z))$를 0에 가까워지도록(정답을 맞추도록) 학습됨.

$$\min_{\theta} \max_{\phi} \mathbb{E}_x \log(D(x)) + \mathbb{E}_z \log(1 - D(G(z)))$$

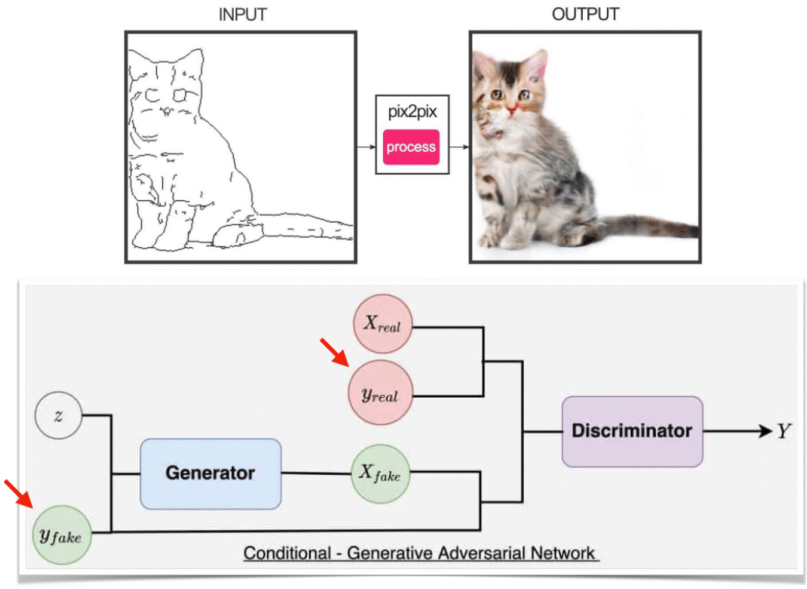
이때 discriminator와 generator는 번갈아 가며 각각 따로 최적화됨.

3. Conditional vs. Unconditional

GAN은 conditional GAN과 unconditional GAN으로 구분될 수 있음.

Unconditional GAN은 generator에 입력으로 넣는 값이 random noise인 GAN임. 반면 Conditional GAN은 generator와 discriminator 각각에 입력 데이터와 함께 해당 데이터에 대한 condition(label 등)을 제공하는 GAN임.

위에 첨부된 사진은 unconditional GAN이고, conditional GAN은 아래와 같은 구조를 가짐.

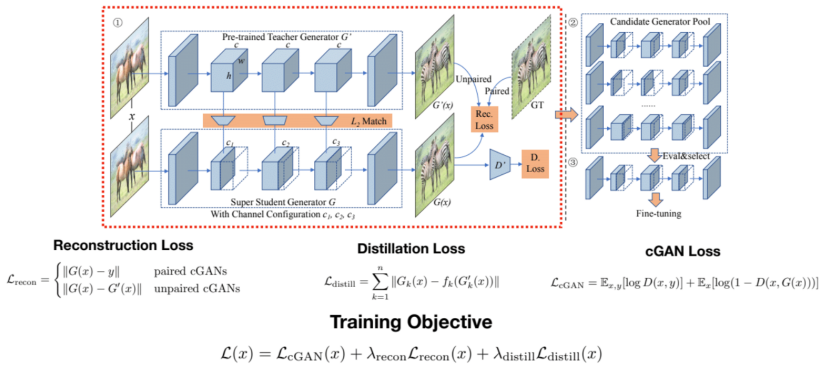


6.1.2. GAN Compression

GAN Compression은 아래와 같이 3가지 loss를 계산해 수행할 수 있음. 이를 통해 generator의 각 channel을 적절히 prune함.

reconstruction loss는 데이터가 paired인 경우 정답과의 비교를, unpaired인 경우 teacher model과의 비교를 pixel-wise하게 계산하는 loss임. distillation loss는 teacher와 student 사이에 대한 distillation을 위한 loss로, 이때 student는 pruning을 적용했으므로 그 차원이 달라 비교 시에 linear projection(f_k)를 사용했음. cGAN(conditional GAN) loss는 student에 대한 GAN loss임.

각 layer를 얼마나 pruning할지는 NAS의 기법을 활용함. generator의 각 layer에 대한 pruning을 각각 다르게 적용한 버전들을 여러 개 준비한 뒤, 최적의 버전을 찾아 fine-tuning한다고 함.

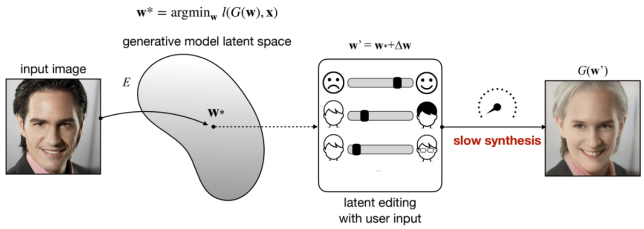


6.1.3. AnyCost GAN

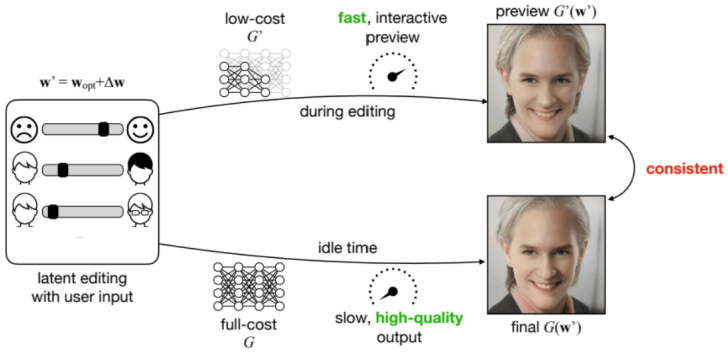
1. AnyCost GAN

AnyCost GAN은 image 생성 시에 우선 더 가벼운 GAN을 활용해 다양한 resolution과 channel 개수로 preview를 생성하고, 이후 무거운 GAN을 활용해 최종 결과물을 확정/생성하는 interactive image editing model임.

GAN을 image editing에 사용할 수 있음. 아래와 같이 input image를 latent space라는 공간에 projection하고, 이후 이 벡터를 generator에 넣어서 image를 생성함. 이때 latent space에서의 벡터 값을 조정하여 image를 edit할 수 있음. 근데 이런 model은 interactive하게 쓰기에 너무 느리다고 함.



AnyCost GAN에서는 더 적은 resolution들과 channel 수들을 지원하는 가벼운 GAN(light-cost GAN)으로 preview를 제공하고, 이후 edit이 확정되면 원본 GAN(full-cost GAN)로 최종 결과물을 도출함.



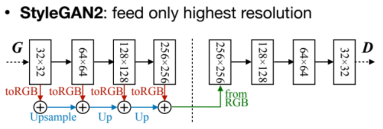
2. *Training Light-cost GAN*

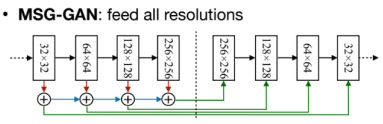
AnyCost GAN에서 활용되는 light-cost GAN은 다양한 resolution과 channel 개수를 처리할 수 있도록 학습됨. 당연하게도 resolution과 channel이 작을수록 더 빠른 계산이 가능함.

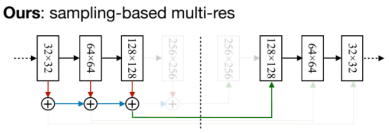
1) Different Resolutions

AnyCost GAN에서는 각 resolution에 대한 image를 독립적으로 생성하여 D(discriminator)에 전달함. 이때 모든 resolution들에 대해 discriminator를 속일 수 있어야 하므로, 학습 시에는 무작위하게 선택된 resolution을 D에 전달함.

StyleGAN2에서는 최대 resolution만을 D에 전달하므로 여러 resolution에 대해 성능이 좋지 못함. MSG-GAN에서는 연산 도중에 도출되는 각 resolution을 모두 D에 전달하는데, 각 resolution들이 서로 종속적으로 계산되므로 성능이 비교적 떨어짐. AnyCost GAN의 방식은 각 resolution에 대해 독립적으로 최적화하므로 computation 비용이 낮고 성능이 좋음.







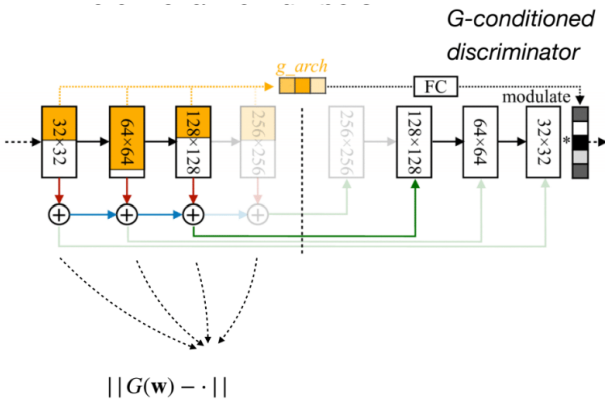
better FID↓ compared to single-resolution models

Resolution	1024	512	256	128	64	32
Single-resolution	3.25	4.17	3.76	4.04	3.32	2.41
MSG-GAN [41]	-	-	4.79	6.34	2.7	3.04
Ours (low res)	-	-	3.49	**3.26**	**2.52**	**2.18**
Ours (high res)	**2.99**	**3.08**	**3.35**	3.98	-	-

여기에서 FID는 실제 image와 생성 image 사이의 거리임.

2) Different Channel Numbers

AnyCost GAN에서는 channel 수를 적응형으로 결정하여 학습함. 이때 단순히 하나의 D가 모든 subset을 처리하도록 하면 일부 subset에서 성능 저하가 발생하는데, 아래와 같이 D에 generator configuration(resolution, channel 수 등)을 전달하여 이를 개선할 수 있음. 이와 같은 처리를 적용한 D를 G-conditioned Discriminator라고 함.



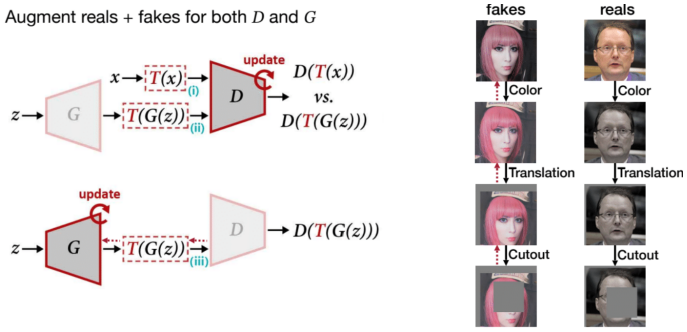
6.1.4. Data-efficient GAN : Differentiable Augmentation

Differentiable Augmentation은 GAN에 대해서 D/G 모두와 real/fake image 모두에 data augmentation을 적용하는 기법임.

GAN은 학습 데이터가 부족한 경우 overfitting이 발생하며 성능이 급격히 떨어짐. 데이터가 부족하면 data augmentation을 적용해 overfitting을 방지할 수 있는데, GAN에서는 real image와 fake image가 활용되므로 모두를 잘 고려한 augmentation을 사용해야 함.

real image와 fake image 중 real image에만 data augmentation(여기에서는 함수 T)를 적용하면 이렇게 바뀐 real image를 타겟해 generated image가 생성되는 문제가 발생함. 또한 D와 G 중 D에 대해서만(D와 G는 번갈아 가며 따로 최적화되므로.) augmentation을 적용해 학습시키면 G에서는 성능이 떨어지는 현상이 발생함.

이에 따라 differentiable augmentation에서는 real/fake image 모두에, 그리고 D와 G 모두에 augmentation을 적용함. 이때 data augmentation으로는 기존 image의 색을 변경하고, shift하고, 일부를 제거하는 등의 기법을 적용함.



6.2. Efficient Video Understanding

video understanding은 cloud와 edge 모두에서 중요한 domain임. video understanding의 특징은 temporal modeling이 필요하다는 것임. 즉, 시간적인 context가 반영되어야 의미 파악이 가능함. 또한 redundancy가 많으므로 efficiency를 개선할 수 있음.

6.2.1. CNN for Video Understanding

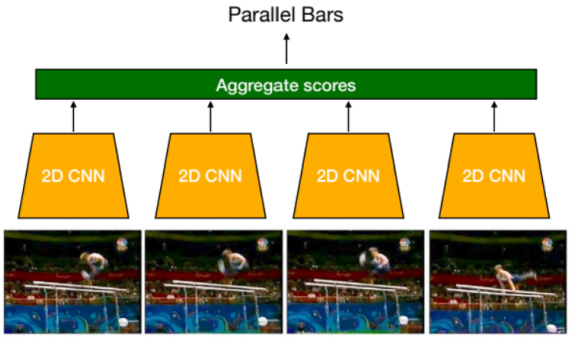
CNN을 활용한 video understanding을 알아보자.

1. 2D CNN

video understanding에 2D CNN을 여러 가지 방법으로 활용할 수 있음. 이때 2D CNN은 computation 비용이 비교적 적다는 장점이 있지만, 성능이 낮거나 정보를 잘 반영하지 못하는 등의 단점이 있음.

1) 단순 aggregation
각 frame에 CNN을 적용한 뒤 단순히 종합해서 예측하는 방법.

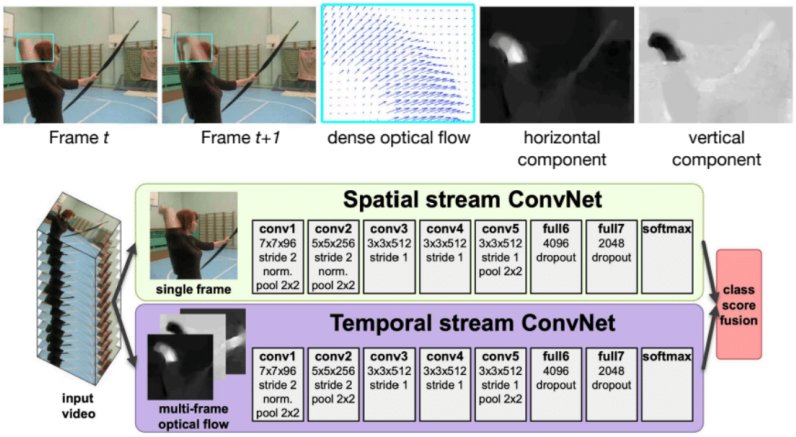
temporal information을 반영하지 못해 성능이 낮음.



2) two stream method
spatial stream과 temporal stream을 활용해 각 frame별로 처리하는 방법.

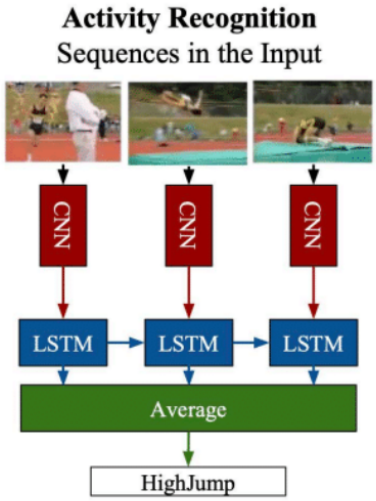
spatial stream에서는 해당 시점의 개별 frame을 사용해 CNN으로 공간적인 처리를 하고, temporal stream에서는 해당 시점에 대한 optical flow를 사용해 CNN으로 시간적인 처리를 함. Optical Flow는 각 pixel에 대해 바로 다음 frame에서 어느 방향으로 이동하는지에 대한 벡터임. 각각의 stream에 대해 연산을 수행한 뒤 이두 stream을 종합함.

optical flow는 추출하는 데에 computation 비용이 많이 들어간다고 함.



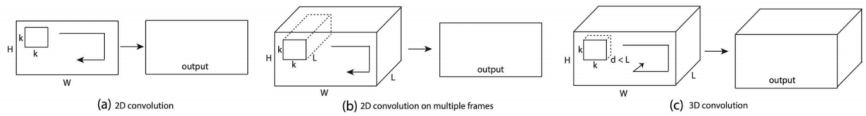
3) CNN and Post-fusion
각 frame에 CNN을 적용한 뒤 post-fusion(LSTM 등) 기법을 사용해 정보를 종합함.

CNN을 적용한 뒤 그 출력을 종합하므로 high level만 전달되고 low level information은 무시됨.



2. 3D CNN

frame을 모아 3D 데이터(실질적으로는 4D 데이터)를 구성한 뒤 3D CNN을 적용하는 것으로 video understanding을 수행할 수 있음.



이때 3D CNN은 더 많은 parameter가 필요한데, I3D(Inflated 3D ConvNet)에서는 pre-trained 2D CNN(ex. ImageNet)의 weight를 가져와 단순히 반복(복사)하여 3D CNN의 kernel을 초기화함. 이를 inflation이라고 함.

3D CNN은 spatiotemporal information을 한 번에 처리하고, low/middle/high information을 모두 활용할 수 있지만, model size, computation 측면에서 비용이 너무 큼.

6.2.2. TSM

1. TSM

TSM(Temporal Shift Module)은 시간 차원에서의 frame간 channel 데이터 shift를 통해 2D CNN만큼의 비용으로 3D CNN만큼의 accuracy 성능을 내는 기법임.

기존의 pre-trained CNN을 활용할 수 있고, 단순히 shift하는 것이므로 별도의 computation이나 parameter의 사용 없이 temporal modeling이 가능함.

이때 얼마나 shift할 것인지는 hyperparameter로 결정함. 너무 적게 shift하면 정보 공유가 잘 이뤄지지 않고, 너무 많이 shift하면 현재 시점에 대한 정보가 부족해짐.

TSM은 구현이 단순하고, online/offline task 모두에 대해 기존의 I3D 등을 활용하는 방식보다 높은 accuracy, 낮은 latency, 높은 throughput, 적은 power, 적은 I/O 및 GPU communication을 보임.

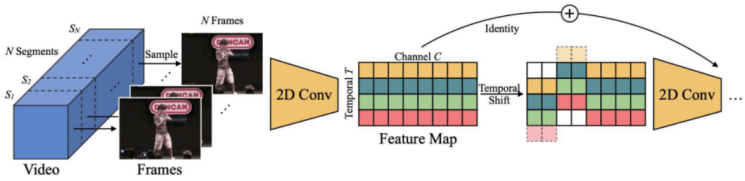
2. Bidirectional vs. Unidirectional

TSM은 bidirectional TSM과 unidirectional TSM으로 나뉨.

1) Bidirectional TSM

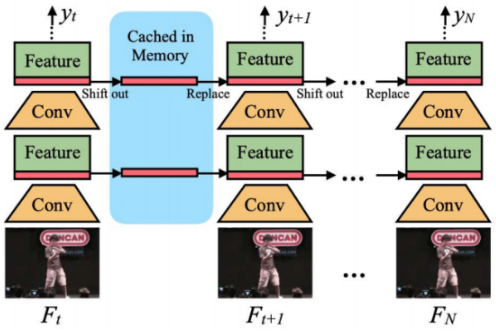
Bidirectional TSM은 offline task에 대한 TSM으로, 과거와 미래의 정보 모두를 활용함.

이때 shift 연산에 의해 맨 처음과 마지막 정보가 손실될 수 있는데, bypass layer에 의해 보상됨.



2) Unidirecitonal TSM
unidirecitonal TSM은 online(real-time) task에 대한 TSM으로, 과거의 정보만을 활용함.

이전 시점의 데이터를 memory에 저장해두고 활용함.



6.3. Efficient Point Cloud Understanding

3D Point cloud는 3D sensor 등으로 수집한 3차원 point들의 집합으로, 각 point는 특정 차원의 feature를 가짐. 이때 3D 공간을 표현하는 기본 단위를 Voxel이라고 함(2D에서의 pixel과 대응됨.). point cloud는 굉장히 1. sparse하고(물체가 없는 지점은 point가 없으므로), 2. irregular함(GPU 연산에서는 연속적이고 regular하게 저장된 데이터가 유리함.). 또한 이는 VR/AR 기기나 자율주행 자동차 등 리소스가 제한된 edge device에서 주로 사용됨. 이에 따라 efficiency가 잘 고려된 기법이 필요함.

6.3.1. PVCNN/SPVCNN

1. PVCNN
PVCNN(Point-voxel CNN)은 voxel branch와 point branch를 모두 활용해 point cloud understanding을 수행하는 model임.

PVCNN은 computation overhead가 적고, 더 빠르면서 높은 accuracy를 보인다고 함.

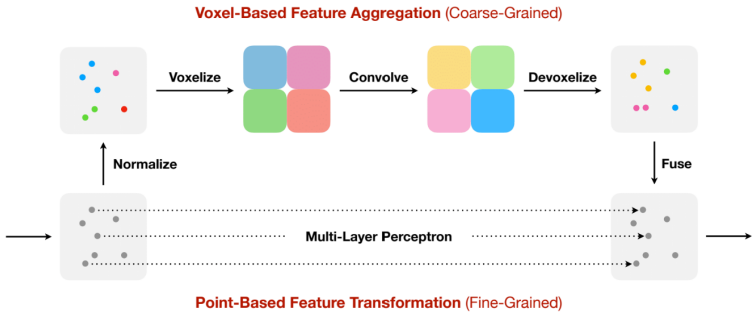
PVCNN에서는 voxel branch와 point branch를 사용함. voxel branch에서는 물체의 넓은 부분이 강조되고, point branch에서는 끝 부분이 강조된다고 함.

1) Voxel Branch
Voxel Branch에서는 point들에 대한 voxelization(3d 공간을 여러 grade의 voxel로 나눔.)을 수행한 뒤 3D convolution을 적용하고, 다시 devoxelization을 수행함. 아래 그림에서는 2D로 표현되었지만 실제로는 3D임.

voxel branch는 point-wise한 연산에 비해 regularity가 높아 더 efficient하고, 인접한 point끼리 정보를 공유하도록 함.

2) Point Branch
Point Branch에서는 point-wise한 MLP를 사용함.

point branch에서는 원본 데이터가 가진 point에 대한 resolution을 유지하고, voxelization에 의한 정보 손실을 복원함.

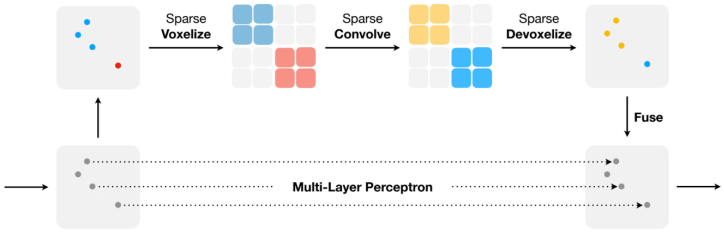


2. SPVCNN

SPVCNN(Sparse PVCNN)은 sparsity를 고려해 기존의 PVCNN에서 efficiency를 개선한 model임.

기존의 PVCNN에서는 voxelization에 의한 정보 손실이 발생할 수 있는데, SPVCNN은 voxelization에 대한 resolution을 적절히 높여 이를 완화할 수 있다고 함.

SPVCNN에서는 아래와 같이 0이 아닌 지점에 대해서만 연산을 수행함. 또한 sparsity를 얼마나 적용할지는 NAS로 최적의 지점을 찾을 수도 있다고 함. 자세한 내용은 논문을 읽어보자.



6.3.2. BEVFusion

BEVFusion(Bird Eye View Fusion)은 camera, LiDAR(레이저를 사용해 주변 정보를 측정하는 기술.) 등으로 얻은 vision, point cloud 데이터를 bird eye view 형태로 합치는 기법임.

여러 기기로부터 얻은 각 데이터는 큰 정보 손실 없이 하나의 공유 공간으로 변환될 수 있어야 함께 활용할 수 있음. BEVFusion에서는 각각을 branch로 처리한 뒤 그 결과를 합쳐 특정 task에 활용함.